

BOOK 6: ADVANCED DEX DEVELOPMENT

Building decentralized exchanges from the ground up. Understanding AMMs, liquidity pools, price discovery, flash loans, and aggregation. This book takes you from DEX user to DEX builder.

CHAPTER 1: DEX ARCHITECTURE OVERVIEW

Decentralized exchanges revolutionized cryptocurrency trading. No intermediaries. No custody. Pure peer-to-peer exchange through smart contracts. Uniswap, Sushiswap, Curve, Balancer - each processes billions in volume. This chapter breaks down how they work at the architectural level.

SECTION 1: CENTRALIZED VS DECENTRALIZED

Traditional exchanges hold your funds. They match orders in private databases. You trust them not to steal, not to be hacked, not to freeze your account. History is full of failures: Mt. Gox, FTX, countless others.

Decentralized exchanges flip this model:

Centralized Exchange (CEX):

- You deposit funds to exchange wallets
- Exchange holds custody
- Order book matching happens off-chain
- Exchange controls execution
- You withdraw when permitted
- Single point of failure

Decentralized Exchange (DEX):

- Funds stay in your wallet until trade
- Smart contracts hold liquidity
- Automated Market Makers price assets
- Code executes deterministically

- Permissionless access
- No single point of failure

The trade-offs are real:

CEX Advantages:

- Faster execution (milliseconds)
- Lower fees (no gas costs)
- More order types (limit, stop, etc.)
- Higher liquidity for major pairs
- Fiat on/off ramps

DEX Advantages:

- Self-custody (your keys, your coins)
- No KYC requirements
- Global access
- Transparent pricing
- Composable with other protocols
- Censorship resistant

SECTION 2: DEX ARCHITECTURE MODELS

DEXes come in several flavors. Each optimizes for different trade-offs.

Order Book DEX:

Order book DEXes mirror traditional exchanges. Buyers and sellers post orders. A matching engine pairs them.

On-Chain Order Book:

- All orders stored on blockchain
- Every order creation/cancellation costs gas
- Slow and expensive
- Examples: Early EtherDelta

Off-Chain Order Book:

- Orders stored on centralized servers
- Only settlement on-chain

- Faster but requires trust
- Examples: 0x Protocol, dYdX

Order Book Structure:

```
class OrderBook:
```

```
    def __init__(self, base_token, quote_token):
```

```
        self.base_token = base_token
```

```
        self.quote_token = quote_token
```

```
        self.bids = []
```

```
        self.asks = []
```

```
    def add_order(self, order):
```

```
        if order.side == "buy":
```

```
            self.bids.append(order)
```

```
            self.bids.sort(key=lambda x: x.price, reverse=True)
```

```
        else:
```

```
            self.asks.append(order)
```

```
            self.asks.sort(key=lambda x: x.price)
```

```
    def match_orders(self):
```

```
        matches = []
```

```
        while self.bids and self.asks:
```

```
            best_bid = self.bids[0]
```

```
            best_ask = self.asks[0]
```

```
            if best_bid.price >= best_ask.price:
```

```
                fill_amount = min(best_bid.remaining, best_ask.remaining)
```

```
                fill_price = best_ask.price
```

```
                matches.append({
```

```
                    "buyer": best_bid.maker,
```

```
                    "seller": best_ask.maker,
```

```
                    "amount": fill_amount,
```

```
                    "price": fill_price
```

```
                })
```

```
                best_bid.remaining -= fill_amount
```

```
best_ask.remaining -= fill_amount
```

```
if best_bid.remaining == 0:  
    self.bids.pop(0)  
if best_ask.remaining == 0:  
    self.asks.pop(0)  
else:  
    break
```

```
return matches
```

Automated Market Maker (AMM):

AMMs eliminated the need for order matching. Smart contracts hold liquidity pools. Mathematical formulas determine prices.

No order book needed. No matching engine. Just math.

Constant Product Market Maker:

$$x * y = k$$

x = reserve of token A

y = reserve of token B

k = constant product (invariant)

When you trade, the product must remain constant:

Before trade: $100 \text{ ETH} * 200,000 \text{ USDC} = 20,000,000$

After buying 1 ETH: $99 \text{ ETH} * ? \text{ USDC} = 20,000,000$

New USDC reserve: $20,000,000 / 99 = 202,020.20$

USDC paid: $202,020.20 - 200,000 = 2,020.20$

Simple AMM Implementation:

```
class ConstantProductAMM:
```

```
    def __init__(self, reserve_a, reserve_b):  
        self.reserve_a = reserve_a
```

```
self.reserve_b = reserve_b
self.k = reserve_a * reserve_b
```

```
def get_price(self):
    return self.reserve_b / self.reserve_a
```

```
def get_amount_out(self, amount_in, token_in):
```

```
    if token_in == "A":
        reserve_in = self.reserve_a
        reserve_out = self.reserve_b
    else:
        reserve_in = self.reserve_b
        reserve_out = self.reserve_a
```

```
    amount_in_with_fee = amount_in * 997
    numerator = amount_in_with_fee * reserve_out
    denominator = (reserve_in * 1000) + amount_in_with_fee
    amount_out = numerator // denominator
```

```
    return amount_out
```

```
def swap(self, amount_in, token_in, min_amount_out):
    amount_out = self.get_amount_out(amount_in, token_in)
```

```
    if amount_out < min_amount_out:
        raise ValueError("Insufficient output amount")
```

```
    if token_in == "A":
        self.reserve_a += amount_in
        self.reserve_b -= amount_out
    else:
        self.reserve_b += amount_in
        self.reserve_a -= amount_out
```

```
    assert self.reserve_a * self.reserve_b >= self.k
```

```
    return amount_out
```

Hybrid Models:

Some DEXes combine approaches:

Curve: Uses different invariant curves optimized for stablecoins

Balancer: Multi-asset pools with custom weights

Uniswap v3: Concentrated liquidity (virtual order book)

DODO: Proactive market maker with oracle pricing

SECTION 3: LIQUIDITY POOLS

Liquidity pools are the heart of AMM DEXes. Users deposit token pairs. Traders swap against the pool.

Pool Mechanics:

```
class LiquidityPool:
```

```
    def __init__(self, token_a, token_b):
```

```
        self.token_a = token_a
```

```
        self.token_b = token_b
```

```
        self.reserve_a = 0
```

```
        self.reserve_b = 0
```

```
        self.total_supply = 0
```

```
        self.lp_balances = {}
```

```
    def add_liquidity(self, provider, amount_a, amount_b):
```

```
        if self.total_supply == 0:
```

```
            liquidity = (amount_a * amount_b) ** 0.5
```

```
        else:
```

```
            liquidity_a = amount_a * self.total_supply // self.reserve_a
```

```
            liquidity_b = amount_b * self.total_supply // self.reserve_b
```

```
            liquidity = min(liquidity_a, liquidity_b)
```

```
            self.reserve_a += amount_a
```

```
            self.reserve_b += amount_b
```

```
            self.total_supply += liquidity
```

```
        if provider not in self.lp_balances:
```

```
            self.lp_balances[provider] = 0
```

```

self.lp_balances[provider] += liquidity

return liquidity

def remove_liquidity(self, provider, liquidity):
    if self.lp_balances.get(provider, 0) < liquidity:
        raise ValueError("Insufficient LP balance")

    amount_a = liquidity * self.reserve_a // self.total_supply
    amount_b = liquidity * self.reserve_b // self.total_supply

    self.reserve_a -= amount_a
    self.reserve_b -= amount_b
    self.total_supply -= liquidity
    self.lp_balances[provider] -= liquidity

    return amount_a, amount_b

def get_lp_value(self, provider):
    lp_balance = self.lp_balances.get(provider, 0)

    if self.total_supply == 0:
        return 0, 0

    share = lp_balance / self.total_supply
    value_a = self.reserve_a * share
    value_b = self.reserve_b * share

    return value_a, value_b

```

LP Tokens:

When you provide liquidity, you receive LP tokens. These represent your share of the pool.

LP tokens are themselves ERC-20 tokens:

- Transferable
- Can be staked for rewards
- Can be used as collateral

- Represent proportional ownership

If pool has 100 ETH and 200,000 USDC, and 1000 LP tokens exist:

- Each LP token represents 0.1 ETH and 200 USDC
- If pool grows to 110 ETH and 220,000 USDC:
- Each LP token now represents 0.11 ETH and 220 USDC

Liquidity providers earn fees:

```
class FeeLiquidityPool(LiquidityPool):
```

```
    def __init__(self, token_a, token_b, fee_percent = 0.3):
        super().__init__(token_a, token_b)
        self.fee_percent = fee_percent
```

```
    def swap(self, amount_in, token_in, min_amount_out):
        fee = amount_in * self.fee_percent / 100
        amount_in_after_fee = amount_in - fee
```

```
        if token_in == self.token_a:
            reserve_in = self.reserve_a
            reserve_out = self.reserve_b
        else:
            reserve_in = self.reserve_b
            reserve_out = self.reserve_a
```

```
        k = reserve_in * reserve_out
```

```
        new_reserve_in = reserve_in + amount_in_after_fee
        new_reserve_out = k / new_reserve_in
        amount_out = reserve_out - new_reserve_out
```

```
        if amount_out < min_amount_out:
            raise ValueError("Insufficient output")
```

```
        if token_in == self.token_a:
            self.reserve_a += amount_in
            self.reserve_b -= amount_out
        else:
            self.reserve_b += amount_in
```



```
self.reserve_a -= amount_out
```

```
return amount_out
```

SECTION 4: FACTORY AND ROUTER PATTERN

Real DEXes use factory contracts to create pools and router contracts to execute swaps.

Factory Contract:

The factory creates new trading pairs. One factory, unlimited pools.

```
class UniswapFactory:
```

```
    def __init__(self):
```

```
        self.pairs = {}
```

```
        self.all_pairs = []
```

```
    def create_pair(self, token_a, token_b):
```

```
        if token_a > token_b:
```

```
            token_a, token_b = token_b, token_a
```

```
            pair_key = (token_a, token_b)
```

```
            if pair_key in self.pairs:
```

```
                raise ValueError("Pair already exists")
```

```
            pair_address = self.deploy_pair(token_a, token_b)
```

```
            self.pairs[pair_key] = pair_address
```

```
            self.all_pairs.append(pair_address)
```

```
            return pair_address
```

```
    def get_pair(self, token_a, token_b):
```

```
        if token_a > token_b:
```

```
            token_a, token_b = token_b, token_a
```

```
return self.pairs.get((token_a, token_b))
```

```
def deploy_pair(self, token_a, token_b):  
    return f'pair_{token_a}_{token_b}'
```

Router Contract:

The router handles complex swaps. Multi-hop paths. Slippage protection. Deadline enforcement.

```
class UniswapRouter:
```

```
    def __init__(self, factory, weth):  
        self.factory = factory  
        self.weth = weth
```

```
    def swap_exact_tokens_for_tokens(  
        self,  
        amount_in,  
        amount_out_min,  
        path,  
        recipient,  
        deadline  
    ):  
        if self.get_current_time() > deadline:  
            raise ValueError("Transaction expired")
```

```
        amounts = self.get_amounts_out(amount_in, path)
```

```
        if amounts[-1] < amount_out_min:  
            raise ValueError("Insufficient output amount")
```

```
        self.transfer_from(path[0], msg_sender, self.get_pair(path[0],  
path[1]), amounts[0])
```

```
        self._swap(amounts, path, recipient)
```

```
        return amounts
```

```

def swap_exact_eth_for_tokens(
    self,
    amount_out_min,
    path,
    recipient,
    deadline,
    value
):
    if path[0] != self.weth:
        raise ValueError("Invalid path")

    if self.get_current_time() > deadline:
        raise ValueError("Transaction expired")

    amounts = self.get_amounts_out(value, path)

    if amounts[-1] < amount_out_min:
        raise ValueError("Insufficient output amount")

    self.wrap_eth(value)
    self.transfer(self.weth, self.get_pair(path[0], path[1]), amounts[0])

    self.swap(amounts, path, recipient)

    return amounts

def _swap(self, amounts, path, recipient):
    for i in range(len(path) - 1):
        token_in = path[i]
        token_out = path[i + 1]

        amount_out = amounts[i + 1]

        to = recipient if i == len(path) - 2 else self.get_pair(path[i + 1],
            path[i + 2])

        pair = self.get_pair(token_in, token_out)
        self.pair_swap(pair, token_in, token_out, amount_out, to)

    def get_amounts_out(self, amount_in, path):

```

```
amounts = [amount_in]
```

```
for i in range(len(path) - 1):
```

```
    pair = self.get_pair(path[i], path[i + 1])
```

```
    reserve_in, reserve_out = self.get_reserves(pair, path[i], path[i + 1])
```

```
    amount_out = self.get_amount_out(amounts[i], reserve_in, reserve_out)
```

```
    amounts.append(amount_out)
```

```
return amounts
```

```
def get_amount_out(self, amount_in, reserve_in, reserve_out):
```

```
    amount_in_with_fee = amount_in * 997
```

```
    numerator = amount_in_with_fee * reserve_out
```

```
    denominator = (reserve_in * 1000) + amount_in_with_fee
```

```
    return numerator // denominator
```

```
def get_pair(self, token_a, token_b):
```

```
    return self.factory.get_pair(token_a, token_b)
```

```
def get_reserves(self, pair, token_a, token_b):
```

```
    pass
```

```
def get_current_time(self):
```

```
    import time
```

```
    return int(time.time())
```

```
def transfer_from(self, token, sender, recipient, amount):
```

```
    pass
```

```
def transfer(self, token, recipient, amount):
```

```
    pass
```

```
def wrap_eth(self, amount):
```

```
    pass
```

```
def pair_swap(self, pair, token_in, token_out, amount_out, to):
```

```
    pass
```

SECTION 5: PRICE IMPACT AND SLIPPAGE

Large trades move prices. Understanding price impact is crucial.

Price Impact:

```
class PriceImpactCalculator:
```

```
    def calculate_price_impact(self, reserve_in, reserve_out, amount_in):  
        spot_price = reserve_out / reserve_in
```

```
        amount_out = self.get_amount_out(amount_in, reserve_in,  
        reserve_out)  
        execution_price = amount_out / amount_in
```

```
        price_impact = (spot_price - execution_price) / spot_price * 100
```

```
        return price_impact
```

```
    def get_amount_out(self, amount_in, reserve_in, reserve_out):  
        amount_in_with_fee = amount_in * 997  
        numerator = amount_in_with_fee * reserve_out  
        denominator = (reserve_in * 1000) + amount_in_with_fee  
        return numerator // denominator
```

```
    def find_optimal_trade_size(self, reserve_in, reserve_out,  
    max_impact=1.0):  
        low = 0  
        high = reserve_in
```

```
        while low < high:  
            mid = (low + high) // 2  
            impact = self.calculate_price_impact(reserve_in, reserve_out, mid)
```

```
            if impact <= max_impact:  
                low = mid + 1  
            else:  
                high = mid
```

return low - 1

Slippage Protection:

Slippage is the difference between expected and actual execution price. It happens because of:

- Price impact from your own trade
- Other transactions executing first
- Pool state changing between quote and execution

class SlippageProtection:

```
def calculate_min_output(self, expected_output, slippage_tolerance):  
    slippage_factor = 1 - (slippage_tolerance / 100)  
    return int(expected_output * slippage_factor)
```

```
def calculate_max_input(self, expected_input, slippage_tolerance):  
    slippage_factor = 1 + (slippage_tolerance / 100)  
    return int(expected_input * slippage_factor)
```

```
def validate_execution(self, expected_output, actual_output,  
    slippage_tolerance):  
    min_output = self.calculate_min_output(expected_output,  
    slippage_tolerance)
```

```
    if actual_output < min_output:  
        raise ValueError(  
            f'Slippage exceeded: expected {expected_output}, "  
            f'got {actual_output}, minimum {min_output}"  
        )
```

```
    return True
```

SECTION 6: MULTI-HOP SWAPS

Not all token pairs have direct pools. Multi-hop routes through intermediate tokens.

Route Finding:

```
class RouteFinder:
```

```
    def __init__(self, factory, common_bases):
```

```
        self.factory = factory
```

```
        self.common_bases = common_bases
```

```
    def find_all_routes(self, token_in, token_out, max_hops=3):
```

```
        routes = []
```

```
        direct_pair = self.factory.get_pair(token_in, token_out)
```

```
        if direct_pair:
```

```
            routes.append([token_in, token_out])
```

```
        if max_hops >= 2:
```

```
            for base in self.common_bases:
```

```
                if base == token_in or base == token_out:
```

```
                    continue
```

```
                pair_1 = self.factory.get_pair(token_in, base)
```

```
                pair_2 = self.factory.get_pair(base, token_out)
```

```
                if pair_1 and pair_2:
```

```
                    routes.append([token_in, base, token_out])
```

```
        if max_hops >= 3:
```

```
            for base1 in self.common_bases:
```

```
                for base2 in self.common_bases:
```

```
                    if base1 == base2:
```

```
                        continue
```

```
                    if base1 == token_in or base1 == token_out:
```

```
                        continue
```

```
                    if base2 == token_in or base2 == token_out:
```

```
                        continue
```

```
                    pair_1 = self.factory.get_pair(token_in, base1)
```

```
                    pair_2 = self.factory.get_pair(base1, base2)
```

```
                    pair_3 = self.factory.get_pair(base2, token_out)
```

```
if pair_1 and pair_2 and pair_3:  
    routes.append([token_in, base1, base2, token_out])
```

```
return routes
```

```
def find_best_route(self, token_in, token_out, amount_in):  
    routes = self.find_all_routes(token_in, token_out)
```

```
if not routes:  
    return None
```

```
best_route = None  
best_output = 0
```

```
for route in routes:  
    try:  
        output = self.simulate_route(route, amount_in)  
        if output > best_output:  
            best_output = output  
            best_route = route  
    except Exception:  
        continue
```

```
return best_route, best_output
```

```
def simulate_route(self, route, amount_in):  
    current_amount = amount_in
```

```
    for i in range(len(route) - 1):  
        pair = self.factory.get_pair(route[i], route[i + 1])  
        reserve_in, reserve_out = self.get_reserves(pair, route[i], route[i + 1])
```

```
        current_amount = self.get_amount_out(current_amount, reserve_in,  
        reserve_out)
```

```
    return current_amount
```

```
def get_reserves(self, pair, token_a, token_b):
```


pass

```
def get_amount_out(self, amount_in, reserve_in, reserve_out):
    amount_in_with_fee = amount_in * 997
    numerator = amount_in_with_fee * reserve_out
    denominator = (reserve_in * 1000) + amount_in_with_fee
    return numerator // denominator
```

SECTION 7: DEX INTERACTION WITH WEB3

Interacting with DEXes programmatically.

Uniswap V2 Integration:

```
from web3 import Web3
```

```
class UniswapV2Client:
```

```
    ROUTER_ABI = [
        {
            "name": "swapExactETHForTokens",
            "type": "function",
            "inputs": [
                {"name": "amountOutMin", "type": "uint256"},
                {"name": "path", "type": "address[]"},
                {"name": "to", "type": "address"},
                {"name": "deadline", "type": "uint256"}
            ],
            "outputs": [{"name": "amounts", "type": "uint256[]"}]
        },
        {
            "name": "swapExactTokensForETH",
            "type": "function",
            "inputs": [
                {"name": "amountIn", "type": "uint256"},
                {"name": "amountOutMin", "type": "uint256"},
                {"name": "path", "type": "address[]"},
                {"name": "to", "type": "address"},
                {"name": "deadline", "type": "uint256"}
            ],
            "outputs": [{"name": "amounts", "type": "uint256[]"}]
        }
    ]
```

```

],
"outputs": [{"name": "amounts", "type": "uint256[]"}]
},
{
"name": "getAmountsOut",
"type": "function",
"inputs": [
{"name": "amountIn", "type": "uint256"},
{"name": "path", "type": "address[]"}
],
"outputs": [{"name": "amounts", "type": "uint256[]"}]
}
]

```

```

def __init__(self, web3, router_address, weth_address):
    self.web3 = web3
    self.router = web3.eth.contract(
        address=router_address,
        abi=self.ROUTER_ABI
    )
    self.weth = weth_address

```

```

def get_quote(self, amount_in, path):
    amounts = self.router.functions.getAmountsOut(amount_in,
        path).call()
    return amounts[-1]

```

```

def buy_token_with_eth(self, token_address, eth_amount,
    slippage_percent, wallet):
    path = [self.weth, token_address]

```

```

    amount_out = self.get_quote(eth_amount, path)
    min_amount_out = int(amount_out * (100 - slippage_percent) /
        100)

```

```

deadline = int(time.time()) + 300

```

```

tx = self.router.functions.swapExactETHForTokens(
    min_amount_out,
    path,

```

```
wallet.address,  
deadline  
) .build_transaction({  
    "from": wallet.address,  
    "value": eth_amount,  
    "gas": 300000,  
    "gasPrice": self.web3.eth.gas_price,  
    "nonce": self.web3.eth.get_transaction_count(wallet.address)  
})
```

```
signed_tx = wallet.sign_transaction(tx)  
tx_hash =  
self.web3.eth.send_raw_transaction(signed_tx.rawTransaction)
```

```
return tx_hash.hex()
```

```
def sell_token_for_eth(self, token_address, token_amount,  
    slippage_percent, wallet):  
    path = [token_address, self.weth]
```

```
    amount_out = self.get_quote(token_amount, path)  
    min_amount_out = int(amount_out * (100 - slippage_percent) /  
    100)
```

```
    deadline = int(time.time()) + 300
```

```
    tx = self.router.functions.swapExactTokensForETH(  
        token_address,  
        min_amount_out,  
        path,  
        wallet.address,  
        deadline  
    ) .build_transaction({  
        "from": wallet.address,  
        "gas": 300000,  
        "gasPrice": self.web3.eth.gas_price,  
        "nonce": self.web3.eth.get_transaction_count(wallet.address)  
    })
```

```
    signed_tx = wallet.sign_transaction(tx)
```

```
tx_hash =  
self.web3.eth.send_raw_transaction(signed_tx.rawTransaction)  
  
return tx_hash.hex()
```

SECTION 8: GAS OPTIMIZATION

DEX transactions can be expensive. Every optimization matters.

Gas Optimization Techniques:

```
class GasOptimizedSwap:  
  
    def __init__(self, web3, router):  
        self.web3 = web3  
        self.router = router  
  
    async def estimate_optimal_gas(self, tx_params):  
        estimated = await self.web3.eth.estimate_gas(tx_params)  
  
        buffer = int(estimated * 0.1)  
        return estimated + buffer  
  
    async def get_optimal_gas_price(self):  
        base_fee = self.web3.eth.get_block("latest")["baseFeePerGas"]  
  
        pending_txs = await self.get_pending_pool()  
        priority_fees = [tx.get("maxPriorityFeePerGas", 0) for tx in  
pending_txs]  
  
        if priority_fees:  
            median_priority = sorted(priority_fees)[len(priority_fees) // 2]  
        else:  
            median_priority = self.web3.to_wei(2, "gwei")  
  
        max_fee = base_fee * 2 + median_priority  
  
        return {  
            "maxFeePerGas": max_fee,
```

```

"maxPriorityFeePerGas": median_priority
}

def batch_approvals(self, tokens, spender, amounts):
    multicall_data = []

    for token, amount in zip(tokens, amounts):
        approve_data = self.encode_approve(token, spender, amount)
        multicall_data.append(approve_data)

    return self.execute_multicall(multicall_data)

async def get_pending_pool(self):
    return []

def encode_approve(self, token, spender, amount):
    pass

def execute_multicall(self, data):
    pass

```

CHAPTER SUMMARY

This chapter covered DEX architecture fundamentals:

Centralized exchanges hold custody and match orders off-chain.
Decentralized exchanges use smart contracts for trustless trading.

AMMs replaced order books with mathematical formulas. Constant product ($x*y=k$) is the most common model.

Liquidity pools hold token pairs. Providers deposit tokens and receive LP tokens representing their share.

Factory contracts create new pools. Router contracts handle complex multi-hop swaps with slippage protection.

Price impact increases with trade size relative to pool reserves.
Slippage protection prevents unfavorable execution.

Multi-hop routes find paths through intermediate tokens when direct pairs don't exist.

Web3 integration enables programmatic DEX interaction for trading bots and applications.

Next chapter: Liquidity Pool Mechanics. We dive deeper into how pools work.

CHAPTER 2: LIQUIDITY POOL MECHANICS

Liquidity pools are the engine of decentralized exchanges. Understanding their mechanics is essential for building, using, and profiting from DEXes. This chapter covers pool mathematics, impermanent loss, fee accumulation, and optimal provision strategies.

SECTION 1: CONSTANT PRODUCT FORMULA DEEP DIVE

The constant product formula ($x * y = k$) seems simple. Its implications are profound.

Mathematical Foundation:

```
class ConstantProductMath:
```

```
    @staticmethod
    def calculate_k(reserve_x, reserve_y):
        return reserve_x * reserve_y
```

```
    @staticmethod
    def calculate_spot_price(reserve_x, reserve_y):
        return reserve_y / reserve_x
```

```
    @staticmethod
    def calculate_marginal_price(reserve_x, reserve_y, amount_x):
        k = reserve_x * reserve_y
```

```
        new_reserve_x = reserve_x + amount_x
```

```
new_reserve_y = k / new_reserve_x
```

```
amount_y = reserve_y - new_reserve_y
```

```
return amount_y / amount_x
```

```
@staticmethod
```

```
def calculate_exact_output(amount_in, reserve_in, reserve_out,  
fee_bps=30):
```

```
    fee_multiplier = 10000 - fee_bps
```

```
    amount_in_with_fee = amount_in * fee_multiplier
```

```
    numerator = amount_in_with_fee * reserve_out
```

```
    denominator = (reserve_in * 10000) + amount_in_with_fee
```

```
    return numerator // denominator
```

```
@staticmethod
```

```
def calculate_required_input(amount_out, reserve_in, reserve_out,  
fee_bps=30):
```

```
    fee_multiplier = 10000 - fee_bps
```

```
    numerator = reserve_in * amount_out * 10000
```

```
    denominator = (reserve_out - amount_out) * fee_multiplier
```

```
    return (numerator // denominator) + 1
```

Price Curve Visualization:

The constant product creates a hyperbolic price curve.

```
class PriceCurveAnalysis:
```

```
    def __init__(self, initial_x, initial_y):
```

```
        self.k = initial_x * initial_y
```

```
        self.initial_x = initial_x
```

```
        self.initial_y = initial_y
```

```
    def get_y_for_x(self, x):
```

```
return self.k / x
```

```
def get_price_at_reserve(self, x):  
    y = self.get_y_for_x(x)  
    return y / x
```

```
def get_curve_points(self, x_min, x_max, num_points = 100):  
    points = []
```

```
    step = (x_max - x_min) / num_points
```

```
    for i in range(num_points + 1):  
        x = x_min + (step * i)  
        y = self.get_y_for_x(x)  
        price = y / x  
        points.append({"x": x, "y": y, "price": price})
```

```
    return points
```

```
def calculate_area_under_curve(self, x_start, x_end):  
    import math
```

```
    area = self.k * math.log(x_end / x_start)  
    return area
```

SECTION 2: LIQUIDITY TOKEN MATHEMATICS

LP tokens represent ownership. Their value changes as pools grow.

LP Token System:

```
class LPTokenSystem:
```

```
    def __init__(self):  
        self.total_supply = 0  
        self.reserve_x = 0  
        self.reserve_y = 0  
        self.balances = {}  
        self.minimum_liquidity = 1000
```



```

def mint(self, provider, amount_x, amount_y):
    if self.total_supply == 0:
        from math import sqrt
        liquidity = int(sqrt(amount_x * amount_y)) -
self.minimum_liquidity

self.balances["dead_address"] = self.minimum_liquidity
self.total_supply = self.minimum_liquidity

else:
    liquidity_x = (amount_x * self.total_supply) // self.reserve_x
    liquidity_y = (amount_y * self.total_supply) // self.reserve_y
    liquidity = min(liquidity_x, liquidity_y)

    if liquidity <= 0:
        raise ValueError("Insufficient liquidity minted")

    self.reserve_x += amount_x
    self.reserve_y += amount_y
    self.total_supply += liquidity

    if provider not in self.balances:
        self.balances[provider] = 0
    self.balances[provider] += liquidity

    return liquidity

def burn(self, provider, liquidity):
    if self.balances.get(provider, 0) < liquidity:
        raise ValueError("Insufficient LP balance")

    amount_x = (liquidity * self.reserve_x) // self.total_supply
    amount_y = (liquidity * self.reserve_y) // self.total_supply

    if amount_x == 0 or amount_y == 0:
        raise ValueError("Insufficient liquidity burned")

    self.balances[provider] -= liquidity
    self.total_supply -= liquidity

```

```

self.reserve_x -= amount_x
self.reserve_y -= amount_y

return amount_x, amount_y

def get_share_value(self, provider):
    balance = self.balances.get(provider, 0)

    if self.total_supply == 0:
        return 0, 0

    share = balance / self.total_supply
    value_x = self.reserve_x * share
    value_y = self.reserve_y * share

    return value_x, value_y

def get_share_percentage(self, provider):
    balance = self.balances.get(provider, 0)

    if self.total_supply == 0:
        return 0

    return (balance / self.total_supply) * 100

```

Initial Liquidity Pricing:

The first liquidity provider sets the initial price. This is critical.

```

class InitialLiquidityProvider:

    def __init__(self, pool):
        self.pool = pool

    def calculate_optimal_initial_deposit(self, target_price,
        total_value_usd, token_x_decimals, token_y_decimals):
        amount_y = total_value_usd / 2

        amount_x = amount_y / target_price

```

```

return {
    "token_x": int(amount_x * (10 ** token_x_decimals)),
    "token_y": int(amount_y * (10 ** token_y_decimals)),
    "implied_price": target_price
}

```

```

def validate_initial_price(self, amount_x, amount_y, expected_price,
tolerance_percent = 1):
    actual_price = amount_y / amount_x

```

```

    deviation = abs(actual_price - expected_price) / expected_price *
100

```

```

    return deviation <= tolerance_percent

```

```

def estimate_arbitrage_risk(self, initial_price, market_price):
    if initial_price == market_price:
        return 0

```

```

    price_diff = abs(initial_price - market_price) / market_price
    return price_diff * 100

```

SECTION 3: IMPERMANENT LOSS

Impermanent loss is the cost of providing liquidity when prices diverge. Every LP must understand it.

Impermanent Loss Calculator:

```

class ImpermanentLossCalculator:

```

```

    @staticmethod
    def calculate_il(price_ratio):
        from math import sqrt

```

```

        r = price_ratio

```

```

        il = (2 * sqrt(r) / (1 + r)) - 1

```

```
return il * 100
```

```
@staticmethod
```

```
def calculate_il_from_prices(initial_price, current_price):
```

```
    price_ratio = current_price / initial_price
```

```
    return ImpermanentLossCalculator.calculate_il(price_ratio)
```

```
@staticmethod
```

```
def calculate_break_even_fees(price_ratio, current_fee_percent):
```

```
    il = ImpermanentLossCalculator.calculate_il(price_ratio)
```

```
    break_even_apy = -il
```

```
    time_to_break_even = break_even_apy / current_fee_percent
```

```
    return {
```

```
        "impermanent_loss_percent": il,
```

```
        "break_even_fee_percent": break_even_apy,
```

```
        "time_to_break_even_at_current_apy": time_to_break_even
```

```
    }
```

```
@staticmethod
```

```
def compare_strategies(initial_investment, initial_price,
```

```
current_price, fee_earned_percent):
```

```
    hold_value = initial_investment / 2 * (1 + current_price /  
initial_price)
```

```
    price_ratio = current_price / initial_price
```

```
    il = ImpermanentLossCalculator.calculate_il(price_ratio)
```

```
    lp_value_before_fees = hold_value * (1 + il / 100)
```

```
    lp_value_after_fees = lp_value_before_fees * (1 +  
fee_earned_percent / 100)
```

```
    return {
```

```
        "hold_value": hold_value,
```

```
        "lp_value_before_fees": lp_value_before_fees,
```

```
        "lp_value_after_fees": lp_value_after_fees,
```

```
        "impermanent_loss_percent": il,
```

```

"fees_earned_percent": fee_earned_percent,
"net_gain_vs_hold": (lp_value_after_fees - hold_value) / hold_value *
100
}

```

Impermanent Loss Table:

```
price_changes = [0.5, 0.75, 1.0, 1.25, 1.5, 2.0, 3.0, 4.0, 5.0]
```

```
il_table = ""
```

```
Price Change  Impermanent Loss
```

```

-----
0.5x (50%)  -5.72%
0.75x (75%) -1.03%
1.0x (100%) 0.00%
1.25x (125%) -0.62%
1.5x (150%) -2.02%
2.0x (200%) -5.72%
3.0x (300%) -13.40%
4.0x (400%) -20.00%
5.0x (500%) -25.46%
""

```

SECTION 4: FEE ACCUMULATION

Fees compensate for impermanent loss. Understanding fee dynamics is essential.

Fee Accumulator:

```
class FeeAccumulator:
```

```

    def __init__(self, fee_percent=0.3):
        self.fee_percent = fee_percent
        self.total_fees_x = 0
        self.total_fees_y = 0
        self.volume_x = 0
        self.volume_y = 0

```

```

def record_swap(self, amount_in, token_in):
    fee = amount_in * self.fee_percent / 100

    if token_in == "x":
        self.total_fees_x += fee
        self.volume_x += amount_in
    else:
        self.total_fees_y += fee
        self.volume_y += amount_in

def calculate_apr(self, reserve_x, reserve_y, days_elapsed):
    total_value = reserve_x + reserve_y

    total_fees = self.total_fees_x + self.total_fees_y

    daily_return = total_fees / total_value / days_elapsed

    apr = ((1 + daily_return) ** 365 - 1) * 100

    return apr

def project_earnings(self, lp_share, reserve_x, reserve_y,
projected_daily_volume, days):
    daily_fees = projected_daily_volume * self.fee_percent / 100

    total_fees = daily_fees * days

    user_fees = total_fees * lp_share

    return user_fees

```

Fee Tiers and Strategies:

```
class FeeTierAnalyzer:
```

```

    def __init__(self):
        self.tiers = {
            "stable": 0.01,

```

```
"standard": 0.3,  
"volatile": 1.0  
}
```

```
def recommend_tier(self, token_a, token_b, volatility_score):  
is_stablecoin_pair = self.check_stablecoin_pair(token_a, token_b)
```

```
if is_stablecoin_pair:  
return "stable", self.tiers["stable"]
```

```
if volatility_score > 0.7:  
return "volatile", self.tiers["volatile"]
```

```
return "standard", self.tiers["standard"]
```

```
def calculate_optimal_fee(self, volatility, typical_trade_size,  
pool_tvl):  
base_fee = 0.3
```

```
volatility_adjustment = volatility * 0.5
```

```
size_ratio = typical_trade_size / pool_tvl  
size_adjustment = size_ratio * 10
```

```
optimal_fee = base_fee + volatility_adjustment + size_adjustment
```

```
optimal_fee = max(0.01, min(optimal_fee, 1.0))
```

```
return optimal_fee
```

```
def check_stablecoin_pair(self, token_a, token_b):  
stablecoins = ["USDC", "USDT", "DAI", "BUSD", "FRAX"]  
return token_a in stablecoins and token_b in stablecoins
```

SECTION 5: ADDING AND REMOVING LIQUIDITY

Practical implementation of liquidity operations.

Liquidity Manager:

```
class LiquidityManager:
```

```
    def __init__(self, pool):  
        self.pool = pool
```

```
    def calculate_add_liquidity_amounts(self, desired_x, desired_y,  
    min_x, min_y):  
        if self.pool.reserve_x == 0 and self.pool.reserve_y == 0:  
            return desired_x, desired_y
```

```
        ratio = self.pool.reserve_y / self.pool.reserve_x
```

```
        optimal_y = desired_x * ratio  
        optimal_x = desired_y / ratio
```

```
        if optimal_y <= desired_y:  
            actual_x = desired_x  
            actual_y = optimal_y  
        else:  
            actual_x = optimal_x  
            actual_y = desired_y
```

```
        if actual_x < min_x or actual_y < min_y:  
            raise ValueError("Amounts below minimum")
```

```
        return int(actual_x), int(actual_y)
```

```
    def estimate_lp_tokens(self, amount_x, amount_y):  
        if self.pool.total_supply == 0:  
            from math import sqrt  
            return int(sqrt(amount_x * amount_y)) - 1000
```

```
        liquidity_x = amount_x * self.pool.total_supply //  
        self.pool.reserve_x  
        liquidity_y = amount_y * self.pool.total_supply //  
        self.pool.reserve_y
```

```
        return min(liquidity_x, liquidity_y)
```



```

def estimate_remove_liquidity_amounts(self, liquidity):
if self.pool.total_supply == 0:
return 0, 0

amount_x = liquidity * self.pool.reserve_x // self.pool.total_supply
amount_y = liquidity * self.pool.reserve_y // self.pool.total_supply

return amount_x, amount_y

def calculate_single_sided_deposit(self, amount, token_in):
half_amount = amount // 2

if token_in == "x":
amount_out = self.pool.get_amount_out(half_amount, "x")
return half_amount, amount_out
else:
amount_out = self.pool.get_amount_out(half_amount, "y")
return amount_out, half_amount

```

Add Liquidity with Web3:

```

class Web3LiquidityManager:

ROUTER_ABI = [
{
"name": "addLiquidity",
"type": "function",
"inputs": [
{"name": "tokenA", "type": "address"},
{"name": "tokenB", "type": "address"},
{"name": "amountADesired", "type": "uint256"},
{"name": "amountBDesired", "type": "uint256"},
{"name": "amountAMin", "type": "uint256"},
{"name": "amountBMin", "type": "uint256"},
{"name": "to", "type": "address"},
{"name": "deadline", "type": "uint256"}
],
"outputs": [
{"name": "amountA", "type": "uint256"},

```

```

{"name": "amountB", "type": "uint256"},
{"name": "liquidity", "type": "uint256"}
],
},
{
"name": "addLiquidityETH",
"type": "function",
"inputs": [
{"name": "token", "type": "address"},
{"name": "amountTokenDesired", "type": "uint256"},
{"name": "amountTokenMin", "type": "uint256"},
{"name": "amountETHMin", "type": "uint256"},
{"name": "to", "type": "address"},
{"name": "deadline", "type": "uint256"}
],
"outputs": [
{"name": "amountToken", "type": "uint256"},
{"name": "amountETH", "type": "uint256"},
{"name": "liquidity", "type": "uint256"}
]
},
{
"name": "removeLiquidity",
"type": "function",
"inputs": [
{"name": "tokenA", "type": "address"},
{"name": "tokenB", "type": "address"},
{"name": "liquidity", "type": "uint256"},
{"name": "amountAMin", "type": "uint256"},
{"name": "amountBMin", "type": "uint256"},
{"name": "to", "type": "address"},
{"name": "deadline", "type": "uint256"}
],
"outputs": [
{"name": "amountA", "type": "uint256"},
{"name": "amountB", "type": "uint256"}
]
}
]

```

```
def __init__(self, web3, router_address):
    self.web3 = web3
    self.router = web3.eth.contract(address = router_address,
abi = self.ROUTER_ABI)
```

```
async def add_liquidity(
    self,
    token_a,
    token_b,
    amount_a,
    amount_b,
    slippage_percent,
    wallet
):
    min_a = int(amount_a * (100 - slippage_percent) / 100)
    min_b = int(amount_b * (100 - slippage_percent) / 100)
    deadline = int(time.time()) + 1200
```

```
tx = self.router.functions.addLiquidity(
    token_a,
    token_b,
    amount_a,
    amount_b,
    min_a,
    min_b,
    wallet.address,
    deadline
).build_transaction({
    "from": wallet.address,
    "gas": 350000,
    "gasPrice": self.web3.eth.gas_price,
    "nonce": self.web3.eth.get_transaction_count(wallet.address)
})
```

```
signed = wallet.sign_transaction(tx)
tx_hash =
self.web3.eth.send_raw_transaction(signed.rawTransaction)

return tx_hash.hex()
```

```

async def add_liquidity_eth(
    self,
    token,
    token_amount,
    eth_amount,
    slippage_percent,
    wallet
):
    min_token = int(token_amount * (100 - slippage_percent) / 100)
    min_eth = int(eth_amount * (100 - slippage_percent) / 100)
    deadline = int(time.time()) + 1200

    tx = self.router.functions.addLiquidityETH(
        token,
        token_amount,
        min_token,
        min_eth,
        wallet.address,
        deadline
    ).build_transaction({
        "from": wallet.address,
        "value": eth_amount,
        "gas": 350000,
        "gasPrice": self.web3.eth.gas_price,
        "nonce": self.web3.eth.get_transaction_count(wallet.address)
    })

    signed = wallet.sign_transaction(tx)
    tx_hash =
self.web3.eth.send_raw_transaction(signed.rawTransaction)

    return tx_hash.hex()

```

SECTION 6: LIQUIDITY POSITION TRACKING

Track positions, calculate returns, monitor performance.

Position Tracker:

class LiquidityPositionTracker:

```
def __init__(self, database):  
    self.db = database
```

```
    async def record_position(self, user_id, pool_address, lp_tokens,  
amount_x, amount_y, price_at_entry):  
        position_id = self.generate_id()
```

```
        await self.db.execute("""  
INSERT INTO lp_positions  
(id, user_id, pool_address, lp_tokens, initial_x, initial_y,  
entry_price, created_at, status)  
VALUES ($1, $2, $3, $4, $5, $6, $7, NOW(), 'active')  
""", position_id, user_id, pool_address, lp_tokens, amount_x,  
amount_y, price_at_entry)
```

```
    return position_id
```

```
    async def close_position(self, position_id, amount_x, amount_y,  
fees_earned_x, fees_earned_y):  
        position = await self.get_position(position_id)
```

```
        il = self.calculate_impermanent_loss(  
position["initial_x"],  
position["initial_y"],  
amount_x,  
amount_y  
)
```

```
        total_return_x = amount_x - position["initial_x"]  
        total_return_y = amount_y - position["initial_y"]
```

```
        await self.db.execute("""  
UPDATE lp_positions SET  
status = 'closed',  
final_x = $1,  
final_y = $2,  
fees_x = $3,  
fees_y = $4,
```

```
impermanent_loss = $5,  
closed_at = NOW()  
WHERE id = $6  
""", amount_x, amount_y, fees_earned_x, fees_earned_y, il,  
position_id)
```

```
async def get_position(self, position_id):  
    row = await self.db.fetchrow(  
        "SELECT * FROM lp_positions WHERE id = $1",  
        position_id  
    )  
    return dict(row) if row else None
```

```
async def get_user_positions(self, user_id, status="active"):  
    rows = await self.db.fetch("""  
    SELECT * FROM lp_positions  
    WHERE user_id = $1 AND status = $2  
    ORDER BY created_at DESC  
    """, user_id, status)
```

```
    return [dict(row) for row in rows]
```

```
async def calculate_current_value(self, position, current_reserve_x,  
current_reserve_y, total_lp_supply):  
    share = position["lp_tokens"] / total_lp_supply
```

```
    current_x = current_reserve_x * share  
    current_y = current_reserve_y * share
```

```
    return current_x, current_y
```

```
def calculate_impermanent_loss(self, initial_x, initial_y, current_x,  
current_y):  
    initial_value = initial_x + initial_y  
    current_value = current_x + current_y
```

```
    hold_value = initial_x * (current_x / initial_x) + initial_y
```

```
    if hold_value == 0:  
        return 0
```

```
il = (current_value - hold_value) / hold_value * 100
return il
```

```
def generate_id(self):
import uuid
return str(uuid.uuid4())[12]
```

SECTION 7: REBALANCING STRATEGIES

Optimal liquidity provision requires active management.

Rebalancing Manager:

```
class RebalancingManager:
```

```
def __init__(self, pool_client, price_oracle):
self.pool = pool_client
self.oracle = price_oracle
```

```
async def check_rebalance_needed(self, position,
threshold_percent = 5):
current_price = await self.oracle.get_price()
entry_price = position["entry_price"]
```

```
price_change = abs(current_price - entry_price) / entry_price * 100
```

```
return price_change > threshold_percent
```

```
async def calculate_rebalance_amounts(self, position,
target_ratio = 0.5):
current_x, current_y = await self.get_current_amounts(position)
```

```
total_value = current_x + current_y
```

```
target_x = total_value * target_ratio
target_y = total_value * (1 - target_ratio)
```

```
delta_x = target_x - current_x
```

```
delta_y = target_y - current_y
```

```
return {  
    "current_x": current_x,  
    "current_y": current_y,  
    "target_x": target_x,  
    "target_y": target_y,  
    "swap_x": delta_x if delta_x > 0 else 0,  
    "swap_y": delta_y if delta_y > 0 else 0  
}
```

```
async def execute_rebalance(self, position, wallet):  
    amounts = await self.calculate_rebalance_amounts(position)
```

```
    await self.remove_liquidity(position, wallet)
```

```
    if amounts["swap_x"] > 0:  
        await self.pool.swap(amounts["swap_y"], "y", wallet)  
    elif amounts["swap_y"] > 0:  
        await self.pool.swap(amounts["swap_x"], "x", wallet)
```

```
    new_position = await self.add_liquidity(  
        amounts["target_x"],  
        amounts["target_y"],  
        wallet  
    )
```

```
    return new_position
```

```
async def get_current_amounts(self, position):  
    pass
```

```
async def remove_liquidity(self, position, wallet):  
    pass
```

```
async def add_liquidity(self, amount_x, amount_y, wallet):  
    pass
```

SECTION 8: ADVANCED POOL TYPES

Beyond constant product, other pool types optimize for specific use cases.

Stable Pool (Curve-style):

```
class StablePool:
```

```
    def __init__(self, reserves, amplification = 100):
```

```
        self.reserves = reserves
```

```
        self.A = amplification
```

```
        self.n = len(reserves)
```

```
    def get_d(self):
```

```
        s = sum(self.reserves)
```

```
        if s == 0:
```

```
            return 0
```

```
        d = s
```

```
        ann = self.A * self.n
```

```
        for _ in range(255):
```

```
            d_p = d
```

```
            for reserve in self.reserves:
```

```
                d_p = d_p * d // (reserve * self.n)
```

```
            d_prev = d
```

```
            d = (ann * s + d_p * self.n) * d // ((ann - 1) * d + (self.n + 1) * d_p)
```

```
            d_p)
```

```
            if abs(d - d_prev) <= 1:
```

```
                return d
```

```
        raise ValueError("D calculation did not converge")
```

```
    def get_y(self, i, j, x):
```

```
        d = self.get_d()
```

```
        ann = self.A * self.n
```

```
c = d
s = 0
```

```
for k in range(self.n):
    if k == i:
        xp = x
    elif k != j:
        xp = self.reserves[k]
    else:
        continue
```

```
s += xp
c = c * d // (xp * self.n)
```

```
c = c * d // (ann * self.n)
b = s + d // ann
```

```
y = d
```

```
for _ in range(255):
    y_prev = y
    y = (y * y + c) // (2 * y + b - d)
```

```
if abs(y - y_prev) <= 1:
    return y
```

```
raise ValueError("Y calculation did not converge")
```

```
def exchange(self, i, j, dx):
    x = self.reserves[i] + dx
    y = self.get_y(i, j, x)
    dy = self.reserves[j] - y
```

```
fee = dy * 4 // 10000
dy -= fee
```

```
self.reserves[i] = x
self.reserves[j] = y + fee
```

```
return dy
```

Weighted Pool (Balancer-style):

```
class WeightedPool:
```

```
    def __init__(self, reserves, weights, swap_fee = 0.003):
        self.reserves = reserves
        self.weights = weights
        self.swap_fee = swap_fee
```

```
    def get_spot_price(self, token_in, token_out):
        balance_in = self.reserves[token_in]
        balance_out = self.reserves[token_out]
        weight_in = self.weights[token_in]
        weight_out = self.weights[token_out]
```

```
    return (balance_in / weight_in) / (balance_out / weight_out)
```

```
    def calculate_out_given_in(self, token_in, token_out, amount_in):
        balance_in = self.reserves[token_in]
        balance_out = self.reserves[token_out]
        weight_in = self.weights[token_in]
        weight_out = self.weights[token_out]
```

```
    amount_in_after_fee = amount_in * (1 - self.swap_fee)
```

```
    weight_ratio = weight_in / weight_out
    balance_ratio = balance_in / (balance_in + amount_in_after_fee)
```

```
    amount_out = balance_out * (1 - balance_ratio ** weight_ratio)
```

```
    return amount_out
```

```
    def swap(self, token_in, token_out, amount_in, min_amount_out):
        amount_out = self.calculate_out_given_in(token_in, token_out,
        amount_in)
```

```
    if amount_out < min_amount_out:
```

```
raise ValueError("Insufficient output amount")
```

```
self.reserves[token_in] += amount_in  
self.reserves[token_out] -= amount_out
```

```
return amount_out
```

CHAPTER SUMMARY

This chapter covered liquidity pool mechanics in depth:

The constant product formula ($x*y=k$) creates a hyperbolic price curve. Every trade moves along this curve.

LP tokens represent proportional ownership of pool reserves. Their value changes with pool growth and trading fees.

Impermanent loss is the opportunity cost of providing liquidity versus holding. It increases with price divergence.

Fee accumulation compensates for impermanent loss. Higher fees and volume can overcome IL.

Adding and removing liquidity requires maintaining the pool ratio. Single-sided deposits require internal swaps.

Position tracking enables performance measurement. Track entry points, current value, fees earned, and IL.

Rebalancing strategies optimize returns by adjusting positions based on price movements.

Advanced pool types (Curve, Balancer) use different mathematical models optimized for specific token types.

Next chapter: Automated Market Makers. We explore AMM theory and implementation.

CHAPTER 3: AUTOMATED MARKET MAKERS

AMMs replaced traditional order books with mathematical formulas. No market makers needed. No order matching. Just pure algorithmic pricing. This chapter explores AMM theory, different curve types, and how to build your own AMM from scratch.

SECTION 1: AMM FUNDAMENTALS

Traditional markets require market makers - entities that quote buy and sell prices. They profit from the spread. They provide liquidity. Without them, markets don't function.

AMMs automate this entirely. Smart contracts hold reserves. Algorithms calculate prices. Anyone can trade. Anyone can provide liquidity.

Core AMM Properties:

Deterministic Pricing: Same inputs always produce same outputs

Continuous Liquidity: Always available to trade (within reserves)

Permissionless: No approval needed to trade or provide liquidity

Transparent: All reserves and formulas visible on-chain

Basic AMM Interface:

```
from abc import ABC, abstractmethod
```

```
class AMM(ABC):
```

```
    @abstractmethod
```

```
    def get_spot_price(self, token_in, token_out):
```

```
        pass
```

```
    @abstractmethod
```

```
    def get_amount_out(self, amount_in, token_in, token_out):
```

```
        pass
```

```
    @abstractmethod
```

```
    def get_amount_in(self, amount_out, token_in, token_out):
```

```
        pass
```

```
@abstractmethod
def swap(self, amount_in, token_in, token_out, min_amount_out):
    pass
```

```
@abstractmethod
def add_liquidity(self, amounts):
    pass
```

```
@abstractmethod
def remove_liquidity(self, lp_amount):
    pass
```

SECTION 2: CONSTANT PRODUCT AMM (UNISWAP V2)

The most influential AMM design. Simple yet powerful.

The Formula:

$$x * y = k$$

Where:

- x is the reserve of token X
- y is the reserve of token Y
- k is the invariant (constant product)

When trading:

- Trader adds dx of token X
- Trader receives dy of token Y
- New reserves: $(x + dx) * (y - dy) = k$

Solving for dy:

$$dy = y - k / (x + dx)$$

$$dy = y - (x * y) / (x + dx)$$

$$dy = y * dx / (x + dx)$$

Complete Uniswap V2 Style AMM:

```
class UniswapV2AMM:
```

```

def __init__(self, token_x, token_y, fee_bps=30):
    self.token_x = token_x
    self.token_y = token_y
    self.reserve_x = 0
    self.reserve_y = 0
    self.fee_bps = fee_bps
    self.total_supply = 0
    self.balances = {}
    self.k_last = 0
    self.minimum_liquidity = 1000

def get_reserves(self):
    return self.reserve_x, self.reserve_y

def get_spot_price(self, token_in, token_out):
    if token_in == self.token_x:
        return self.reserve_y / self.reserve_x if self.reserve_x > 0 else 0
    else:
        return self.reserve_x / self.reserve_y if self.reserve_y > 0 else 0

def get_amount_out(self, amount_in, token_in, token_out):
    if amount_in <= 0:
        raise ValueError("Insufficient input amount")

    if token_in == self.token_x:
        reserve_in = self.reserve_x
        reserve_out = self.reserve_y
    else:
        reserve_in = self.reserve_y
        reserve_out = self.reserve_x

    if reserve_in <= 0 or reserve_out <= 0:
        raise ValueError("Insufficient liquidity")

    amount_in_with_fee = amount_in * (10000 - self.fee_bps)
    numerator = amount_in_with_fee * reserve_out
    denominator = (reserve_in * 10000) + amount_in_with_fee

    return numerator // denominator

```

```

def get_amount_in(self, amount_out, token_in, token_out):
    if amount_out <= 0:
        raise ValueError("Insufficient output amount")

    if token_out == self.token_x:
        reserve_in = self.reserve_y
        reserve_out = self.reserve_x
    else:
        reserve_in = self.reserve_x
        reserve_out = self.reserve_y

    if reserve_in <= 0 or reserve_out <= 0:
        raise ValueError("Insufficient liquidity")

    if amount_out >= reserve_out:
        raise ValueError("Insufficient liquidity")

    numerator = reserve_in * amount_out * 10000
    denominator = (reserve_out - amount_out) * (10000 - self.fee_bps)

    return (numerator // denominator) + 1

def swap(self, amount_in, token_in, min_amount_out, to):
    if token_in == self.token_x:
        token_out = self.token_y
    else:
        token_out = self.token_x

    amount_out = self.get_amount_out(amount_in, token_in, token_out)

    if amount_out < min_amount_out:
        raise ValueError("Insufficient output amount")

    if token_in == self.token_x:
        self.reserve_x += amount_in
        self.reserve_y -= amount_out
    else:
        self.reserve_y += amount_in
        self.reserve_x -= amount_out

```



```
new_k = self.reserve_x * self.reserve_y
old_k = (self.reserve_x - (amount_in if token_in == self.token_x
else -amount_out)) * \
(self.reserve_y - (-amount_out if token_in == self.token_x else
amount_in))
```

```
assert new_k >= old_k, "K decreased"
```

```
return amount_out
```

```
def mint(self, amount_x, amount_y, to):
    if self.total_supply == 0:
        from math import sqrt
        liquidity = int(sqrt(amount_x * amount_y)) -
self.minimum_liquidity
```

```
self.balances["0x0"] = self.minimum_liquidity
self.total_supply = self.minimum_liquidity
else:
    liquidity_x = amount_x * self.total_supply // self.reserve_x
    liquidity_y = amount_y * self.total_supply // self.reserve_y
    liquidity = min(liquidity_x, liquidity_y)
```

```
if liquidity <= 0:
    raise ValueError("Insufficient liquidity minted")
```

```
self.reserve_x += amount_x
self.reserve_y += amount_y
```

```
if to not in self.balances:
    self.balances[to] = 0
    self.balances[to] += liquidity
    self.total_supply += liquidity
```

```
self.k_last = self.reserve_x * self.reserve_y
```

```
return liquidity
```

```
def burn(self, liquidity, to):
```

```
balance = self.balances.get(to, 0)
```

```
if balance < liquidity:  
    raise ValueError("Insufficient balance")
```

```
amount_x = liquidity * self.reserve_x // self.total_supply  
amount_y = liquidity * self.reserve_y // self.total_supply
```

```
if amount_x <= 0 or amount_y <= 0:  
    raise ValueError("Insufficient liquidity burned")
```

```
self.balances[to] -= liquidity  
self.total_supply -= liquidity
```

```
self.reserve_x -= amount_x  
self.reserve_y -= amount_y
```

```
self.k_last = self.reserve_x * self.reserve_y
```

```
return amount_x, amount_y
```

```
def get_price_impact(self, amount_in, token_in):  
    spot_price = self.get_spot_price(token_in, self.token_y if token_in  
    == self.token_x else self.token_x)
```

```
    amount_out = self.get_amount_out(amount_in, token_in,  
    self.token_y if token_in == self.token_x else self.token_x)  
    execution_price = amount_out / amount_in
```

```
    impact = (spot_price - execution_price) / spot_price * 100
```

```
    return impact
```

SECTION 3: CONSTANT SUM AMM

The simplest AMM. Fixed 1:1 exchange rate. Used for pegged assets.

$$x + y = k$$

Constant Sum Implementation:

```
class ConstantSumAMM:
```

```
    def __init__(self, token_x, token_y, fee_bps=5):
        self.token_x = token_x
        self.token_y = token_y
        self.reserve_x = 0
        self.reserve_y = 0
        self.fee_bps = fee_bps
```

```
    def get_spot_price(self, token_in, token_out):
        return 1.0
```

```
    def get_amount_out(self, amount_in, token_in, token_out):
        fee = amount_in * self.fee_bps // 10000
        amount_out = amount_in - fee
```

```
        if token_out == self.token_x:
            if amount_out > self.reserve_x:
                raise ValueError("Insufficient liquidity")
            else:
                if amount_out > self.reserve_y:
                    raise ValueError("Insufficient liquidity")
```

```
        return amount_out
```

```
    def swap(self, amount_in, token_in, min_amount_out, to):
        if token_in == self.token_x:
            token_out = self.token_y
        else:
            token_out = self.token_x
```

```
        amount_out = self.get_amount_out(amount_in, token_in, token_out)
```

```
        if amount_out < min_amount_out:
            raise ValueError("Insufficient output amount")
```

```
        if token_in == self.token_x:
            self.reserve_x += amount_in
```

```

self.reserve_y -= amount_out
else:
self.reserve_y += amount_in
self.reserve_x -= amount_out

return amount_out

```

Problem with Constant Sum:

Constant sum AMMs are vulnerable to arbitrage drain. If external price differs from 1:1, arbitrageurs will drain one side completely.

External price: $1 X = 1.1 Y$

AMM price: $1 X = 1 Y$

Arbitrageur buys all X from AMM at 1:1, sells externally at 1:1.1.
Pool drained of X.

SECTION 4: STABLESWAP AMM (CURVE)

Curve's innovation: blend constant sum and constant product. Tight prices near equilibrium, infinite liquidity protection at extremes.

The StableSwap Invariant:

$$A^n \cdot \sum(x_i) + D = A^n \cdot D + D^{(n+1)} / (n^n \cdot \prod(x_i))$$

Where:

- A is the amplification coefficient
- n is the number of tokens
- D is the invariant (total value at equilibrium)
- x_i are the token reserves

StableSwap Implementation:

```
class StableSwapAMM:
```

```
    def __init__(self, tokens, amplification=100, fee_bps=4):
```

```
self.tokens = tokens
self.n = len(tokens)
self.reserves = {token: 0 for token in tokens}
self.A = amplification
self.fee_bps = fee_bps
self.total_supply = 0
self.balances = {}
```

```
def get_d(self, reserves=None):
    if reserves is None:
        reserves = [self.reserves[t] for t in self.tokens]
```

```
s = sum(reserves)
```

```
if s == 0:
    return 0
```

```
d = s
ann = self.A * self.n
```

```
for _ in range(255):
    d_p = d
```

```
for x in reserves:
    d_p = d_p * d // (x * self.n + 1)
```

```
d_prev = d
numerator = (ann * s + d_p * self.n) * d
denominator = (ann - 1) * d + (self.n + 1) * d_p
d = numerator // denominator
```

```
if abs(d - d_prev) <= 1:
    return d
```

```
raise ValueError("D did not converge")
```

```
def get_y(self, i, j, x_new):
    reserves = [self.reserves[t] for t in self.tokens]
```

```
d = self.get_d(reserves)
```

```
ann = self.A * self.n
```

```
c = d
```

```
s = 0
```

```
for k in range(self.n):
```

```
    if k == i:
```

```
        x = x_new
```

```
    elif k == j:
```

```
        continue
```

```
    else:
```

```
        x = reserves[k]
```

```
s += x
```

```
c = c * d // (x * self.n)
```

```
c = c * d // (ann * self.n)
```

```
b = s + d // ann
```

```
y = d
```

```
for _ in range(255):
```

```
    y_prev = y
```

```
    y = (y * y + c) // (2 * y + b - d)
```

```
    if abs(y - y_prev) <= 1:
```

```
        return y
```

```
raise ValueError("Y did not converge")
```

```
def get_amount_out(self, amount_in, token_in, token_out):
```

```
    i = self.tokens.index(token_in)
```

```
    j = self.tokens.index(token_out)
```

```
    x_new = self.reserves[token_in] + amount_in
```

```
    y_new = self.get_y(i, j, x_new)
```

```
    dy = self.reserves[token_out] - y_new - 1
```

```

fee = dy * self.fee_bps // 10000

return dy - fee

def swap(self, amount_in, token_in, token_out, min_amount_out, to):
    amount_out = self.get_amount_out(amount_in, token_in, token_out)

    if amount_out < min_amount_out:
        raise ValueError("Insufficient output amount")

    self.reserves[token_in] += amount_in
    self.reserves[token_out] -= amount_out

    return amount_out

def add_liquidity(self, amounts, to):
    d0 = self.get_d() if self.total_supply > 0 else 0

    for token, amount in amounts.items():
        self.reserves[token] += amount

    d1 = self.get_d()

    if self.total_supply == 0:
        liquidity = d1
    else:
        liquidity = (d1 - d0) * self.total_supply // d0

    if to not in self.balances:
        self.balances[to] = 0
    self.balances[to] += liquidity
    self.total_supply += liquidity

    return liquidity

def remove_liquidity(self, liquidity, to):
    amounts = {}

    for token in self.tokens:
        amount = liquidity * self.reserves[token] // self.total_supply

```

```

amounts[token] = amount
self.reserves[token] -= amount

self.balances[to] -= liquidity
self.total_supply -= liquidity

return amounts

def get_virtual_price(self):
    d = self.get_d()

    if self.total_supply == 0:
        return 1

    return d * 10**18 // self.total_supply

```

SECTION 5: WEIGHTED POOLS (BALANCER)

Balancer generalized constant product to arbitrary weights. Not just 50/50 pools.

The Weighted Invariant:

$$\text{prod}(B_i^{W_i}) = k$$

Where:

- B_i is the balance of token i
- W_i is the weight of token i (sum to 1)
- k is the invariant

Weighted Pool Implementation:

```

class WeightedPoolAMM:

    def __init__(self, tokens, weights, fee_bps=30):
        if abs(sum(weights.values()) - 1.0) > 0.0001:
            raise ValueError("Weights must sum to 1")

        self.tokens = tokens

```



```
self.weights = weights
self.reserves = {token: 0 for token in tokens}
self.fee_bps = fee_bps
self.total_supply = 0
self.balances = {}
```

```
def get_invariant(self):
    result = 1
```

```
    for token in self.tokens:
        balance = self.reserves[token]
        weight = self.weights[token]
```

```
    if balance == 0:
        return 0
```

```
    result *= balance ** weight
```

```
    return result
```

```
def get_spot_price(self, token_in, token_out):
    balance_in = self.reserves[token_in]
    balance_out = self.reserves[token_out]
    weight_in = self.weights[token_in]
    weight_out = self.weights[token_out]
```

```
    return (balance_in / weight_in) / (balance_out / weight_out)
```

```
def get_amount_out(self, amount_in, token_in, token_out):
    balance_in = self.reserves[token_in]
    balance_out = self.reserves[token_out]
    weight_in = self.weights[token_in]
    weight_out = self.weights[token_out]
```

```
    amount_in_after_fee = amount_in * (10000 - self.fee_bps) // 10000
```

```
    weight_ratio = weight_in / weight_out
```

```
    new_balance_in = balance_in + amount_in_after_fee
    balance_ratio = balance_in / new_balance_in
```

```
amount_out = balance_out * (1 - balance_ratio ** weight_ratio)
```

```
return int(amount_out)
```

```
def get_amount_in(self, amount_out, token_in, token_out):
```

```
    balance_in = self.reserves[token_in]
```

```
    balance_out = self.reserves[token_out]
```

```
    weight_in = self.weights[token_in]
```

```
    weight_out = self.weights[token_out]
```

```
    weight_ratio = weight_out / weight_in
```

```
    new_balance_out = balance_out - amount_out
```

```
    balance_ratio = balance_out / new_balance_out
```

```
    amount_in_before_fee = balance_in * (balance_ratio ** weight_ratio  
- 1)
```

```
    amount_in = amount_in_before_fee * 10000 // (10000 -  
self.fee_bps)
```

```
    return int(amount_in) + 1
```

```
def swap(self, amount_in, token_in, token_out, min_amount_out, to):
```

```
    amount_out = self.get_amount_out(amount_in, token_in, token_out)
```

```
    if amount_out < min_amount_out:
```

```
        raise ValueError("Insufficient output amount")
```

```
    self.reserves[token_in] += amount_in
```

```
    self.reserves[token_out] -= amount_out
```

```
    return amount_out
```

```
def join_pool(self, amounts, to):
```

```
    if self.total_supply == 0:
```

```
        invariant = 1
```

```
    for token in self.tokens:
```

```
        self.reserves[token] = amounts[token]
```

```
invariant *= amounts[token] ** self.weights[token]
```

```
liquidity = int(invariant)
```

```
else:
```

```
ratio = float("inf")
```

```
for token in self.tokens:
```

```
token_ratio = amounts[token] / self.reserves[token]
```

```
ratio = min(ratio, token_ratio)
```

```
liquidity = int(self.total_supply * ratio)
```

```
for token in self.tokens:
```

```
self.reserves[token] += int(self.reserves[token] * ratio)
```

```
if to not in self.balances:
```

```
self.balances[to] = 0
```

```
self.balances[to] += liquidity
```

```
self.total_supply += liquidity
```

```
return liquidity
```

```
def exit_pool(self, liquidity, to):
```

```
ratio = liquidity / self.total_supply
```

```
amounts = {}
```

```
for token in self.tokens:
```

```
amount = int(self.reserves[token] * ratio)
```

```
amounts[token] = amount
```

```
self.reserves[token] -= amount
```

```
self.balances[to] -= liquidity
```

```
self.total_supply -= liquidity
```

```
return amounts
```

SECTION 6: CONCENTRATED LIQUIDITY (UNISWAP V3)

Uniswap V3 revolutionized AMMs with concentrated liquidity. LPs

provide liquidity within price ranges.

Concentrated Liquidity Concepts:

Traditional AMM: Liquidity spread across infinite price range (0 to infinity)

Concentrated Liquidity: Liquidity focused on specific price ranges

Benefits:

- Capital efficient (same liquidity, less capital)
- Higher fees for LPs in active ranges
- Custom risk profiles

Trade-offs:

- More complex position management
- Liquidity can go out of range
- Higher impermanent loss risk in range

Simplified Concentrated Liquidity:

from math import sqrt

class ConcentratedLiquidityPosition:

```
def __init__(self, liquidity, price_lower, price_upper, current_price):
```

```
    self.liquidity = liquidity
```

```
    self.price_lower = price_lower
```

```
    self.price_upper = price_upper
```

```
    self.sqrt_price_lower = sqrt(price_lower)
```

```
    self.sqrt_price_upper = sqrt(price_upper)
```

```
    self.sqrt_current_price = sqrt(current_price)
```

```
def get_amounts(self, current_price):
```

```
    sqrt_p = sqrt(current_price)
```

```
    if current_price < self.price_lower:
```

```
        amount_x = self.liquidity * (1/self.sqrt_price_lower - 1/
```

```
        self.sqrt_price_upper)
```

```
        amount_y = 0
```

```

elif current_price > self.price_upper:
    amount_x = 0
    amount_y = self.liquidity * (self.sqrt_price_upper -
self.sqrt_price_lower)

else:
    amount_x = self.liquidity * (1/sqrt_p - 1/self.sqrt_price_upper)
    amount_y = self.liquidity * (sqrt_p - self.sqrt_price_lower)

return amount_x, amount_y

def is_in_range(self, current_price):
    return self.price_lower <= current_price <= self.price_upper

class ConcentratedLiquidityPool:

    def __init__(self, token_x, token_y, fee_bps=30, tick_spacing=60):
        self.token_x = token_x
        self.token_y = token_y
        self.fee_bps = fee_bps
        self.tick_spacing = tick_spacing

        self.positions = []
        self.current_price = 1.0
        self.liquidity = 0

    def price_to_tick(self, price):
        from math import log

        tick = int(log(price) / log(1.0001))
        return tick - (tick % self.tick_spacing)

    def tick_to_price(self, tick):
        return 1.0001 ** tick

    def add_position(self, owner, amount_x, amount_y, price_lower,
price_upper):
        tick_lower = self.price_to_tick(price_lower)

```

```
tick_upper = self.price_to_tick(price_upper)
```

```
sqrt_lower = sqrt(self.tick_to_price(tick_lower))
```

```
sqrt_upper = sqrt(self.tick_to_price(tick_upper))
```

```
sqrt_current = sqrt(self.current_price)
```

```
if self.current_price < price_lower:
```

```
    liquidity = amount_x * sqrt_lower * sqrt_upper / (sqrt_upper -  
    sqrt_lower)
```

```
elif self.current_price > price_upper:
```

```
    liquidity = amount_y / (sqrt_upper - sqrt_lower)
```

```
else:
```

```
    liquidity_x = amount_x * sqrt_current * sqrt_upper / (sqrt_upper -  
    sqrt_current)
```

```
    liquidity_y = amount_y / (sqrt_current - sqrt_lower)
```

```
    liquidity = min(liquidity_x, liquidity_y)
```

```
position = {
```

```
    "owner": owner,
```

```
    "liquidity": liquidity,
```

```
    "tick_lower": tick_lower,
```

```
    "tick_upper": tick_upper,
```

```
    "fees_x": 0,
```

```
    "fees_y": 0
```

```
}
```

```
self.positions.append(position)
```

```
if tick_lower <= self.price_to_tick(self.current_price) < tick_upper:
```

```
    self.liquidity += liquidity
```

```
return len(self.positions) - 1
```

```
def get_active_liquidity(self):
```

```
    current_tick = self.price_to_tick(self.current_price)
```

```
    total_liquidity = 0
```

```
    for pos in self.positions:
```

```
        if pos["tick_lower"] <= current_tick < pos["tick_upper"]:
```

```
            total_liquidity += pos["liquidity"]
```

```
return total_liquidity
```

```
def swap(self, amount_in, token_in, min_amount_out):  
    remaining = amount_in  
    total_out = 0
```

```
    while remaining > 0:  
        active_liquidity = self.get_active_liquidity()
```

```
    if active_liquidity == 0:  
        break
```

```
    sqrt_current = sqrt(self.current_price)
```

```
    if token_in == self.token_x:  
        next_tick = self.get_next_initialized_tick_down()  
        sqrt_next = sqrt(self.tick_to_price(next_tick))
```

```
    max_amount_in = active_liquidity * (1/sqrt_next - 1/sqrt_current)
```

```
    if remaining <= max_amount_in:  
        new_sqrt_price = 1 / (1/sqrt_current + remaining/active_liquidity)  
        amount_out = active_liquidity * (sqrt_current - new_sqrt_price)
```

```
    self.current_price = new_sqrt_price ** 2  
    remaining = 0  
    else:
```

```
        amount_out = active_liquidity * (sqrt_current - sqrt_next)
```

```
    self.current_price = self.tick_to_price(next_tick)  
    remaining -= max_amount_in
```

```
    else:  
        next_tick = self.get_next_initialized_tick_up()  
        sqrt_next = sqrt(self.tick_to_price(next_tick))
```

```
    max_amount_in = active_liquidity * (sqrt_next - sqrt_current)
```

```
    if remaining <= max_amount_in:
```

```

new_sqrt_price = sqrt_current + remaining/active_liquidity
amount_out = active_liquidity * (1/sqrt_current - 1/new_sqrt_price)

self.current_price = new_sqrt_price ** 2
remaining = 0
else:
amount_out = active_liquidity * (1/sqrt_current - 1/sqrt_next)

self.current_price = self.tick_to_price(next_tick)
remaining -= max_amount_in

total_out += amount_out

if total_out < min_amount_out:
raise ValueError("Insufficient output")

return total_out

def get_next_initialized_tick_down(self):
current_tick = self.price_to_tick(self.current_price)

for pos in sorted(self.positions, key=lambda p: p["tick_lower"],
reverse=True):
if pos["tick_lower"] < current_tick:
return pos["tick_lower"]

return current_tick - self.tick_spacing

def get_next_initialized_tick_up(self):
current_tick = self.price_to_tick(self.current_price)

for pos in sorted(self.positions, key=lambda p: p["tick_upper"]):
if pos["tick_upper"] > current_tick:
return pos["tick_upper"]

return current_tick + self.tick_spacing

```

SECTION 7: ORACLE-BASED AMMs (DODO)

DODO introduced Proactive Market Making. External oracles provide prices. AMM concentrates liquidity around oracle price.

PMM Concept:

Traditional AMM: Price discovered through trading

PMM: Price set by oracle, liquidity concentrated

Benefits:

- Less impermanent loss
- Tighter spreads
- More capital efficient

PMM Implementation:

class ProactiveMarketMaker:

```
def __init__(self, base_token, quote_token, oracle, k=0.5):
```

```
    self.base_token = base_token
```

```
    self.quote_token = quote_token
```

```
    self.oracle = oracle
```

```
    self.k = k
```

```
    self.base_reserve = 0
```

```
    self.quote_reserve = 0
```

```
    self.base_target = 0
```

```
    self.quote_target = 0
```

```
def get_oracle_price(self):
```

```
    return self.oracle.get_price(self.base_token, self.quote_token)
```

```
def get_mid_price(self):
```

```
    if self.base_reserve == 0:
```

```
        return self.get_oracle_price()
```

```
    oracle_price = self.get_oracle_price()
```

```
    r = self.base_target / self.base_reserve
```

```
    if r > 1:
```

```
        coefficient = 1 - self.k + self.k * r
```

```

else:
    coefficient = 1

return oracle_price * coefficient

def get_amount_out(self, amount_in, token_in):
    oracle_price = self.get_oracle_price()

    if token_in == self.base_token:
        if self.base_reserve + amount_in <= self.base_target:
            return amount_in * oracle_price

        r = self.base_reserve / self.base_target
        new_r = (self.base_reserve + amount_in) / self.base_target

        integral_old = self._integrate_price_curve(r, oracle_price)
        integral_new = self._integrate_price_curve(new_r, oracle_price)

        return (integral_new - integral_old) * self.base_target

    else:
        if self.quote_reserve + amount_in <= self.quote_target:
            return amount_in / oracle_price

        r = self.quote_reserve / self.quote_target
        new_r = (self.quote_reserve + amount_in) / self.quote_target

        integral_old = self._integrate_price_curve(r, 1/oracle_price)
        integral_new = self._integrate_price_curve(new_r, 1/oracle_price)

        return (integral_new - integral_old) * self.quote_target

def _integrate_price_curve(self, r, base_price):
    if r <= 1:
        return r * base_price
    else:
        linear_part = base_price
        premium_part = base_price * self.k * (r - 1) ** 2 / 2
        return linear_part + premium_part

```

```

def swap(self, amount_in, token_in, min_amount_out):
    amount_out = self.get_amount_out(amount_in, token_in)

    if amount_out < min_amount_out:
        raise ValueError("Insufficient output")

    if token_in == self.base_token:
        self.base_reserve += amount_in
        self.quote_reserve -= amount_out
    else:
        self.quote_reserve += amount_in
        self.base_reserve -= amount_out

    return amount_out

```

SECTION 8: BUILDING A COMPLETE AMM

Putting together all components into a production-ready AMM.

Complete AMM System:

class CompleteAMM:

```

def __init__(self, token_x, token_y, fee_bps=30, protocol_fee_bps=5):
    self.token_x = token_x
    self.token_y = token_y
    self.fee_bps = fee_bps
    self.protocol_fee_bps = protocol_fee_bps

    self.reserve_x = 0
    self.reserve_y = 0
    self.total_supply = 0
    self.balances = {}

    self.protocol_fees_x = 0
    self.protocol_fees_y = 0

    self.cumulative_price_x = 0
    self.cumulative_price_y = 0

```

```
self.last_block_timestamp = 0
```

```
self.minimum_liquidity = 1000
```

```
self.locked = False
```

```
def _update_price_accumulators(self, timestamp):
```

```
time_elapsed = timestamp - self.last_block_timestamp
```

```
if time_elapsed > 0 and self.reserve_x > 0 and self.reserve_y > 0:
```

```
self.cumulative_price_x += (self.reserve_y / self.reserve_x) *
```

```
time_elapsed
```

```
self.cumulative_price_y += (self.reserve_x / self.reserve_y) *
```

```
time_elapsed
```

```
self.last_block_timestamp = timestamp
```

```
def _mint_protocol_fee(self):
```

```
if self.protocol_fee_bps == 0:
```

```
return
```

```
root_k = int((self.reserve_x * self.reserve_y) ** 0.5)
```

```
root_k_last = int((self.protocol_fees_x * self.protocol_fees_y) ** 0.5)
```

```
if self.protocol_fees_x > 0 else 0
```

```
if root_k > root_k_last:
```

```
numerator = self.total_supply * (root_k - root_k_last)
```

```
denominator = root_k * (10000 // self.protocol_fee_bps - 1) +
```

```
root_k_last
```

```
liquidity = numerator // denominator
```

```
if liquidity > 0:
```

```
if "protocol" not in self.balances:
```

```
self.balances["protocol"] = 0
```

```
self.balances["protocol"] += liquidity
```

```
self.total_supply += liquidity
```

```
def swap(self, amount_in, token_in, min_amount_out, to,
```

```
timestamp):
```

```
if self.locked:
```

```
raise ValueError("Reentrancy")
```

```
self.locked = True
```

```
try:  
self._update_price_accumulators(timestamp)
```

```
if token_in == self.token_x:  
    reserve_in = self.reserve_x  
    reserve_out = self.reserve_y  
else:  
    reserve_in = self.reserve_y  
    reserve_out = self.reserve_x
```

```
amount_in_with_fee = amount_in * (10000 - self.fee_bps)  
numerator = amount_in_with_fee * reserve_out  
denominator = reserve_in * 10000 + amount_in_with_fee  
amount_out = numerator // denominator
```

```
if amount_out < min_amount_out:  
    raise ValueError("Insufficient output")
```

```
if token_in == self.token_x:  
    self.reserve_x += amount_in  
    self.reserve_y -= amount_out  
else:  
    self.reserve_y += amount_in  
    self.reserve_x -= amount_out
```

```
return amount_out
```

```
finally:  
self.locked = False
```

```
def mint(self, amount_x, amount_y, to, timestamp):  
    if self.locked:  
        raise ValueError("Reentrancy")  
    self.locked = True
```

```
try:  
    self._update_price_accumulators(timestamp)  
    self._mint_protocol_fee()
```

```

if self.total_supply == 0:
    liquidity = int((amount_x * amount_y) ** 0.5) -
self.minimum_liquidity

self.balances["0x0"] = self.minimum_liquidity
self.total_supply = self.minimum_liquidity
else:
    liquidity_x = amount_x * self.total_supply // self.reserve_x
    liquidity_y = amount_y * self.total_supply // self.reserve_y
    liquidity = min(liquidity_x, liquidity_y)

if liquidity <= 0:
    raise ValueError("Insufficient liquidity")

self.reserve_x += amount_x
self.reserve_y += amount_y

if to not in self.balances:
    self.balances[to] = 0
    self.balances[to] += liquidity
    self.total_supply += liquidity

return liquidity

finally:
    self.locked = False

def burn(self, liquidity, to, timestamp):
    if self.locked:
        raise ValueError("Reentrancy")
    self.locked = True

    try:
        self._update_price_accumulators(timestamp)
        self._mint_protocol_fee()

    if self.balances.get(to, 0) < liquidity:
        raise ValueError("Insufficient balance")

```

```
amount_x = liquidity * self.reserve_x // self.total_supply
amount_y = liquidity * self.reserve_y // self.total_supply
```

```
if amount_x == 0 or amount_y == 0:
    raise ValueError("Insufficient liquidity burned")
```

```
self.balances[to] -= liquidity
self.total_supply -= liquidity
```

```
self.reserve_x -= amount_x
self.reserve_y -= amount_y
```

```
return amount_x, amount_y
```

```
finally:
    self.locked = False
```

```
def get_twap(self, time_start, time_end, price_cumulative_start,
price_cumulative_end):
    time_elapsed = time_end - time_start
```

```
if time_elapsed == 0:
    return self.reserve_y / self.reserve_x
```

```
return (price_cumulative_end - price_cumulative_start) /
time_elapsed
```

CHAPTER SUMMARY

This chapter covered Automated Market Maker theory and implementation:

Constant Product ($x*y=k$) is the foundation of Uniswap-style AMMs. Simple, robust, widely adopted.

Constant Sum ($x+y=k$) provides fixed 1:1 rates but is vulnerable to complete liquidity drain.

StableSwap (Curve) blends constant sum and constant product for

efficient stablecoin trading.

Weighted Pools (Balancer) generalize constant product to arbitrary token weights.

Concentrated Liquidity (Uniswap V3) focuses liquidity in price ranges for capital efficiency.

Oracle-based AMMs (DODO) use external price feeds to reduce impermanent loss.

Production AMMs need reentrancy protection, price accumulators for TWAP, and protocol fee mechanisms.

Next chapter: Price Discovery and Oracles. We explore how DEXes discover and report prices.

CHAPTER 4: PRICE DISCOVERY AND ORACLES

Prices are the lifeblood of DeFi. Every swap, loan, liquidation depends on accurate prices. This chapter covers how DEXes discover prices, how oracles report them, and how to build price feeds that resist manipulation.

SECTION 1: PRICE DISCOVERY MECHANISMS

AMMs discover prices through trading. Every swap moves the price. Arbitrageurs keep DEX prices aligned with global markets.

Price Discovery in Constant Product:

class PriceDiscovery:

```
def __init__(self, reserve_x, reserve_y):
    self.reserve_x = reserve_x
    self.reserve_y = reserve_y
```

```
def get_spot_price(self):
    return self.reserve_y / self.reserve_x
```



```
def get_marginal_price(self):  
    return self.reserve_y / self.reserve_x
```

```
def get_execution_price(self, amount_in, token_in):  
    if token_in == "x":  
        amount_out = self.get_amount_out(amount_in, self.reserve_x,  
self.reserve_y)  
        return amount_out / amount_in  
    else:  
        amount_out = self.get_amount_out(amount_in, self.reserve_y,  
self.reserve_x)  
        return amount_in / amount_out
```

```
def get_amount_out(self, amount_in, reserve_in, reserve_out):  
    amount_in_with_fee = amount_in * 997  
    numerator = amount_in_with_fee * reserve_out  
    denominator = reserve_in * 1000 + amount_in_with_fee  
    return numerator // denominator
```

```
def simulate_arbitrage(self, external_price):  
    current_price = self.get_spot_price()
```

```
    if abs(current_price - external_price) / external_price < 0.003:  
        return None
```

```
    if current_price < external_price:  
        amount_to_buy = self.calculate_arbitrage_amount(external_price,  
"buy")  
        return {"action": "buy_x", "amount": amount_to_buy}  
    else:  
        amount_to_sell = self.calculate_arbitrage_amount(external_price,  
"sell")  
        return {"action": "sell_x", "amount": amount_to_sell}
```

```
def calculate_arbitrage_amount(self, target_price, direction):  
    from math import sqrt
```

```
    k = self.reserve_x * self.reserve_y
```

```
    target_reserve_x = sqrt(k / target_price)
```



```
})
```

```
return sorted(opportunities, key=lambda x: x["profit_estimate"],  
reverse=True)
```

```
def estimate_profit(self, pool, external_price, price_diff):  
    from math import sqrt
```

```
    reserve_x, reserve_y = pool.get_reserves()  
    k = reserve_x * reserve_y
```

```
    target_x = sqrt(k / external_price)  
    delta_x = abs(reserve_x - target_x)
```

```
    if price_diff > 0:  
        amount_out = pool.get_amount_out(delta_x, "x")  
        revenue = amount_out * external_price  
        cost = delta_x  
    else:  
        amount_in = delta_x  
        cost = amount_in * external_price  
        amount_out = pool.get_amount_out(amount_in, "y")  
        revenue = amount_out
```

```
    gas_cost = 0.01  
    profit = revenue - cost - gas_cost
```

```
    return profit
```

SECTION 2: TIME-WEIGHTED AVERAGE PRICE (TWAP)

Spot prices are easily manipulated within a single block. TWAP averages prices over time, making manipulation expensive.

TWAP Oracle:

```
class TWAPOracle:
```

```
    def __init__(self):
```

```

self.price_cumulative = 0
self.last_price = 0
self.last_timestamp = 0
self.observations = []
self.observation_cardinality = 24

def update(self, price, timestamp):
    if self.last_timestamp == 0:
        self.last_timestamp = timestamp
        self.last_price = price
    return

time_elapsed = timestamp - self.last_timestamp

if time_elapsed > 0:
    self.price_cumulative += self.last_price * time_elapsed

self.observations.append({
    "timestamp": timestamp,
    "price_cumulative": self.price_cumulative
})

if len(self.observations) > self.observation_cardinality:
    self.observations.pop(0)

self.last_price = price
self.last_timestamp = timestamp

def get_twap(self, seconds_ago):
    if len(self.observations) < 2:
        return self.last_price

target_timestamp = self.last_timestamp - seconds_ago

observation_old = None
observation_new = self.observations[-1]

for obs in self.observations:
    if obs["timestamp"] <= target_timestamp:
        observation_old = obs

```

```
else:  
    break
```

```
if observation_old is None:  
    observation_old = self.observations[0]
```

```
time_elapsed = observation_new["timestamp"] -  
observation_old["timestamp"]
```

```
if time_elapsed == 0:  
    return self.last_price
```

```
price_diff = observation_new["price_cumulative"] -  
observation_old["price_cumulative"]
```

```
return price_diff / time_elapsed
```

```
def consult(self, period):  
    return self.get_twap(period)
```

Uniswap V2 Style Price Accumulator:

```
class UniswapV2PriceAccumulator:
```

```
    def __init__(self):  
        self.reserve0 = 0  
        self.reserve1 = 0  
        self.price0_cumulative_last = 0  
        self.price1_cumulative_last = 0  
        self.block_timestamp_last = 0
```

```
    def update_reserves(self, reserve0, reserve1, timestamp):  
        time_elapsed = timestamp - self.block_timestamp_last
```

```
        if time_elapsed > 0 and self.reserve0 > 0 and self.reserve1 > 0:  
            price0 = (self.reserve1 << 112) // self.reserve0  
            price1 = (self.reserve0 << 112) // self.reserve1
```

```
        self.price0_cumulative_last += price0 * time_elapsed
```

```
self.price1_cumulative_last += price1 * time_elapsed
```

```
self.reserve0 = reserve0
```

```
self.reserve1 = reserve1
```

```
self.block_timestamp_last = timestamp
```

```
def get_current_cumulative_prices(self, timestamp):
```

```
    price0_cumulative = self.price0_cumulative_last
```

```
    price1_cumulative = self.price1_cumulative_last
```

```
    time_elapsed = timestamp - self.block_timestamp_last
```

```
    if time_elapsed > 0 and self.reserve0 > 0 and self.reserve1 > 0:
```

```
        price0 = (self.reserve1 < 112) // self.reserve0
```

```
        price1 = (self.reserve0 < 112) // self.reserve1
```

```
        price0_cumulative += price0 * time_elapsed
```

```
        price1_cumulative += price1 * time_elapsed
```

```
    return price0_cumulative, price1_cumulative
```

TWAP Consumer:

```
class TWAPConsumer:
```

```
    def __init__(self, oracle_address):
```

```
        self.oracle = oracle_address
```

```
        self.observations = []
```

```
        self.period = 3600
```

```
    def update(self, timestamp):
```

```
        price0_cumulative, price1_cumulative =
```

```
        self.oracle.get_current_cumulative_prices(timestamp)
```

```
        self.observations.append({
```

```
            "timestamp": timestamp,
```

```
            "price0_cumulative": price0_cumulative,
```

```
            "price1_cumulative": price1_cumulative
```

```
        })
```

```

if len(self.observations) > 2:
    self.observations.pop(0)

def consult(self, amount_in, token_in):
    if len(self.observations) < 2:
        raise ValueError("Not enough observations")

    obs_old = self.observations[0]
    obs_new = self.observations[-1]

    time_elapsed = obs_new["timestamp"] - obs_old["timestamp"]

    if time_elapsed == 0:
        raise ValueError("Period not elapsed")

    if token_in == 0:
        price_average = (obs_new["price0_cumulative"] -
obs_old["price0_cumulative"]) // time_elapsed
    else:
        price_average = (obs_new["price1_cumulative"] -
obs_old["price1_cumulative"]) // time_elapsed

    amount_out = (amount_in * price_average) > > 112

    return amount_out

```

SECTION 3: CHAINLINK ORACLES

Chainlink provides decentralized price feeds. Multiple independent nodes report prices. Aggregation produces final value.

Chainlink Price Feed Consumer:

```

class ChainlinkPriceFeed:

    AGGREGATOR_ABI = [
        {
            "name": "latestRoundData",

```

```

"type": "function",
"inputs": [],
"outputs": [
{"name": "roundId", "type": "uint80"},
{"name": "answer", "type": "int256"},
{"name": "startedAt", "type": "uint256"},
{"name": "updatedAt", "type": "uint256"},
{"name": "answeredInRound", "type": "uint80"}
],
},
{
"name": "decimals",
"type": "function",
"inputs": [],
"outputs": [{"name": "", "type": "uint8"}]
}
]

```

```

def __init__(self, web3, feed_address):
    self.web3 = web3
    self.feed = web3.eth.contract(address=feed_address,
abi=self.AGGREGATOR_ABI)

```

```

def get_latest_price(self):
    round_data = self.feed.functions.latestRoundData().call()

```

```

    round_id = round_data[0]
    answer = round_data[1]
    started_at = round_data[2]
    updated_at = round_data[3]
    answered_in_round = round_data[4]

```

```

    if answer <= 0:
        raise ValueError("Invalid price")

```

```

    if updated_at == 0:
        raise ValueError("Round not complete")

```

```

    if answered_in_round < round_id:
        raise ValueError("Stale price")

```



```
return answer
```

```
def get_price_with_staleness_check(self,  
max_staleness_seconds = 3600):  
    import time
```

```
    round_data = self.feed.functions.latestRoundData().call()
```

```
    answer = round_data[1]  
    updated_at = round_data[3]
```

```
    current_time = int(time.time())
```

```
    if current_time - updated_at > max_staleness_seconds:  
        raise ValueError(f"Price stale: {current_time - updated_at} seconds  
old")
```

```
    return answer
```

```
def get_decimals(self):  
    return self.feed.functions.decimals().call()
```

```
def get_price_in_usd(self, amount, token_decimals):  
    price = self.get_latest_price()  
    decimals = self.get_decimals()
```

```
    value = (amount * price) // (10 ** token_decimals)
```

```
    return value / (10 ** decimals)
```

Multi-Feed Price Aggregator:

```
class MultiPriceFeedAggregator:
```

```
    def __init__(self):  
        self.feeds = {}
```

```
    def add_feed(self, token, feed):
```

```

self.feeds[token] = feed

def get_price(self, token):
    if token not in self.feeds:
        raise ValueError(f"No feed for {token}")

    return self.feeds[token].get_latest_price()

def get_price_in_token(self, amount, from_token, to_token):
    from_price = self.get_price(from_token)
    to_price = self.get_price(to_token)

    from_decimals = self.feeds[from_token].get_decimals()
    to_decimals = self.feeds[to_token].get_decimals()

    normalized_from = from_price * (10 ** to_decimals)
    normalized_to = to_price * (10 ** from_decimals)

    return (amount * normalized_from) // normalized_to

def get_usd_value(self, token, amount, token_decimals):
    price = self.get_price(token)
    decimals = self.feeds[token].get_decimals()

    usd_value = (amount * price) // (10 ** token_decimals)

    return usd_value / (10 ** decimals)

```

SECTION 4: ON-CHAIN PRICE CALCULATIONS

Calculate prices directly from DEX reserves.

DEX Price Calculator:

```
class DEXPriceCalculator:
```

```

    PAIR_ABI = [
    {
        "name": "getReserves",

```

```

"type": "function",
"inputs": [],
"outputs": [
{"name": "reserve0", "type": "uint112"},
{"name": "reserve1", "type": "uint112"},
{"name": "blockTimestampLast", "type": "uint32"}
],
},
{
"name": "token0",
"type": "function",
"inputs": [],
"outputs": [{"name": "", "type": "address"}]
},
{
"name": "token1",
"type": "function",
"inputs": [],
"outputs": [{"name": "", "type": "address"}]
}
]

```

```

def __init__(self, web3):
self.web3 = web3

```

```

def get_pair_price(self, pair_address, token_address, token_decimals,
base_decimals):
pair = self.web3.eth.contract(address=pair_address,
abi=self.PAIR_ABI)

```

```

reserves = pair.functions.getReserves().call()
reserve0 = reserves[0]
reserve1 = reserves[1]

```

```

token0 = pair.functions.token0().call()

```

```

if token_address.lower() == token0.lower():
price = (reserve1 * (10 ** token_decimals)) / (reserve0 * (10 **
base_decimals))
else:

```

```
price = (reserve0 * (10 ** token_decimals)) / (reserve1 * (10 **  
base_decimals))
```

```
return price
```

```
def get_token_price_via_path(self, path, pair_addresses,  
decimals_list):  
    price = 1.0
```

```
    for i in range(len(path) - 1):  
        token_in = path[i]  
        pair = pair_addresses[i]
```

```
        pair_price = self.get_pair_price(  
            pair,  
            token_in,  
            decimals_list[i],  
            decimals_list[i + 1]  
        )
```

```
        price *= pair_price
```

```
    return price
```

```
def get_liquidity_depth(self, pair_address):  
    pair = self.web3.eth.contract(address=pair_address,  
abi=self.PAIR_ABI)
```

```
    reserves = pair.functions.getReserves().call()
```

```
    return {  
        "reserve0": reserves[0],  
        "reserve1": reserves[1],  
        "total_liquidity": reserves[0] + reserves[1]  
    }
```

SECTION 5: MANIPULATION RESISTANCE

Price oracles must resist manipulation. Several attack vectors exist.

Common Attack Vectors:

Flash Loan Attack: Borrow large amount, manipulate price, profit, repay loan

Sandwich Attack: Front-run and back-run a large trade

Oracle Delay Exploit: Use stale oracle prices before update

Manipulation Detection:

```
class ManipulationDetector:
```

```
    def __init__(self, oracle, threshold_percent = 5):
        self.oracle = oracle
        self.threshold = threshold_percent
        self.price_history = []
        self.window_size = 10
```

```
    def record_price(self, price, timestamp):
        self.price_history.append({
            "price": price,
            "timestamp": timestamp
        })
```

```
    if len(self.price_history) > self.window_size:
        self.price_history.pop(0)
```

```
    def detect_anomaly(self, current_price):
        if len(self.price_history) < 3:
            return False, None
```

```
        prices = [p["price"] for p in self.price_history]
        avg_price = sum(prices) / len(prices)
```

```
        deviation = abs(current_price - avg_price) / avg_price * 100
```

```
        if deviation > self.threshold:
            return True, {
                "current_price": current_price,
                "average_price": avg_price,
```

```
"deviation_percent": deviation
}
```

```
return False, None
```

```
def calculate_volatility(self):
    if len(self.price_history) < 2:
        return 0
```

```
prices = [p["price"] for p in self.price_history]
```

```
returns = []
for i in range(1, len(prices)):
    ret = (prices[i] - prices[i-1]) / prices[i-1]
    returns.append(ret)
```

```
if len(returns) == 0:
    return 0
```

```
avg_return = sum(returns) / len(returns)
variance = sum((r - avg_return) ** 2 for r in returns) / len(returns)
```

```
return variance ** 0.5
```

Secure Price Oracle:

```
class SecurePriceOracle:
```

```
    def __init__(self, primary_source, backup_sources,
max_deviation=5):
        self.primary = primary_source
        self.backups = backup_sources
        self.max_deviation = max_deviation
        self.last_valid_price = None
        self.last_update_time = 0
```

```
    def get_price(self):
        primary_price = self._safe_get_price(self.primary)
```

```
if primary_price is None:  
    return self._get_backup_price()
```

```
if self._validate_price(primary_price):  
    self.last_valid_price = primary_price  
    return primary_price
```

```
    return self._get_backup_price()
```

```
def _safe_get_price(self, source):  
    try:  
        return source.get_latest_price()  
    except Exception:  
        return None
```

```
def _validate_price(self, price):  
    if price <= 0:  
        return False
```

```
    if self.last_valid_price is not None:  
        deviation = abs(price - self.last_valid_price) / self.last_valid_price *  
        100
```

```
    if deviation > self.max_deviation:  
        return False
```

```
    backup_prices = []  
    for source in self.backups:  
        backup_price = self._safe_get_price(source)  
        if backup_price is not None:  
            backup_prices.append(backup_price)
```

```
    if len(backup_prices) >= 2:  
        median_backup = sorted(backup_prices)[len(backup_prices) // 2]  
        deviation = abs(price - median_backup) / median_backup * 100
```

```
    if deviation > self.max_deviation:  
        return False
```

```
    return True
```

```

def _get_backup_price(self):
    prices = []

    for source in self.backups:
        price = self._safe_get_price(source)
        if price is not None and price > 0:
            prices.append(price)

    if len(prices) == 0:
        if self.last_valid_price is not None:
            return self.last_valid_price
        raise ValueError("No valid price available")

    prices.sort()
    median_price = prices[len(prices) // 2]

    self.last_valid_price = median_price
    return median_price

```

SECTION 6: BUILDING A PRICE FEED

Complete price feed implementation for a DEX.

DEX Price Feed:

```

class DEXPriceFeed:

    def __init__(self, web3, factory_address, weth_address, usdc_address):
        self.web3 = web3
        self.factory = factory_address
        self.weth = weth_address
        self.usdc = usdc_address

        self.price_cache = {}
        self.cache_duration = 60

    async def get_token_price_usd(self, token_address):
        cache_key = token_address.lower()

```



```
if cache_key in self.price_cache:
    cached = self.price_cache[cache_key]
    if time.time() - cached["timestamp"] < self.cache_duration:
        return cached["price"]
```

```
if token_address.lower() == self.usdc.lower():
    return 1.0
```

```
if token_address.lower() == self.weth.lower():
    return await self._get_weth_price()
```

```
price = await self._calculate_token_price(token_address)
```

```
self.price_cache[cache_key] = {
    "price": price,
    "timestamp": time.time()
}
```

```
return price
```

```
async def _get_weth_price(self):
    weth_usdc_pair = await self._get_pair(self.weth, self.usdc)
```

```
if weth_usdc_pair is None:
    raise ValueError("No WETH/USDC pair")
```

```
reserves = await self._get_reserves(weth_usdc_pair)
```

```
token0 = await self._get_token0(weth_usdc_pair)
```

```
if token0.lower() == self.weth.lower():
    price = (reserves[1] * 10**12) / reserves[0]
else:
    price = (reserves[0] * 10**12) / reserves[1]
```

```
return price
```

```
async def _calculate_token_price(self, token_address):
    weth_pair = await self._get_pair(token_address, self.weth)
```

```

if weth_pair:
    price_in_weth = await self._get_price_from_pair(
        weth_pair, token_address, self.weth
    )
    weth_price = await self._get_weth_price()
    return price_in_weth * weth_price

usdc_pair = await self._get_pair(token_address, self.usdc)

if usdc_pair:
    price_in_usdc = await self._get_price_from_pair(
        usdc_pair, token_address, self.usdc
    )
    return price_in_usdc * (10**12)

raise ValueError(f"No price path for {token_address}")

async def _get_price_from_pair(self, pair_address, token_address,
    base_address):
    reserves = await self._get_reserves(pair_address)
    token0 = await self._get_token0(pair_address)

    if token0.lower() == token_address.lower():
        return reserves[1] / reserves[0]
    else:
        return reserves[0] / reserves[1]

async def _get_pair(self, token_a, token_b):
    pass

async def _get_reserves(self, pair_address):
    pass

async def _get_token0(self, pair_address):
    pass

```

SECTION 7: PRICE IMPACT ESTIMATION

Estimate price impact before executing trades.

Price Impact Estimator:

```
class PriceImpactEstimator:
```

```
    def __init__(self, pool_data_provider):  
        self.provider = pool_data_provider
```

```
    async def estimate_price_impact(self, token_in, token_out,  
amount_in):  
        pool = await self.provider.get_pool(token_in, token_out)
```

```
        if pool is None:  
            return None
```

```
        reserves = await self.provider.get_reserves(pool)
```

```
        if token_in == await self.provider.get_token0(pool):  
            reserve_in = reserves[0]  
            reserve_out = reserves[1]  
        else:  
            reserve_in = reserves[1]  
            reserve_out = reserves[0]
```

```
        spot_price = reserve_out / reserve_in
```

```
        amount_out = self._get_amount_out(amount_in, reserve_in,  
reserve_out)  
        execution_price = amount_out / amount_in
```

```
        price_impact = (spot_price - execution_price) / spot_price * 100
```

```
        return {  
            "spot_price": spot_price,  
            "execution_price": execution_price,  
            "price_impact_percent": price_impact,  
            "amount_out": amount_out  
        }
```

```
def _get_amount_out(self, amount_in, reserve_in, reserve_out):  
    amount_in_with_fee = amount_in * 997  
    numerator = amount_in_with_fee * reserve_out  
    denominator = reserve_in * 1000 + amount_in_with_fee  
    return numerator // denominator
```

```
async def find_optimal_trade_size(self, token_in, token_out,  
max_impact_percent):  
    pool = await self.provider.get_pool(token_in, token_out)  
    reserves = await self.provider.get_reserves(pool)
```

```
    if token_in == await self.provider.get_token0(pool):  
        reserve_in = reserves[0]  
        reserve_out = reserves[1]  
    else:  
        reserve_in = reserves[1]  
        reserve_out = reserves[0]
```

```
    low = 0  
    high = reserve_in // 2
```

```
    while high - low > reserve_in // 1000:  
        mid = (low + high) // 2
```

```
    spot = reserve_out / reserve_in  
    amount_out = self._get_amount_out(mid, reserve_in, reserve_out)  
    exec_price = amount_out / mid  
    impact = (spot - exec_price) / spot * 100
```

```
    if impact <= max_impact_percent:  
        low = mid  
    else:  
        high = mid
```

```
    return low
```

```
async def calculate_slippage_tolerance(self, token_in, token_out,  
amount_in, volatility_percent):  
    impact = await self.estimate_price_impact(token_in, token_out,  
amount_in)
```

```

base_slippage = impact["price_impact_percent"]

volatility_buffer = volatility_percent * 0.5

recommended_slippage = base_slippage + volatility_buffer + 0.5

return min(recommended_slippage, 50)

```

SECTION 8: ORACLE INTEGRATION PATTERNS

Common patterns for integrating oracles in DeFi protocols.

Lending Protocol Oracle:

```

class LendingOracleIntegration:

```

```

    def __init__(self, chainlink_feeds, fallback_dex_oracle,
price_deviation_threshold=10):
        self.chainlink = chainlink_feeds
        self.dex_oracle = fallback_dex_oracle
        self.threshold = price_deviation_threshold

```

```

    async def get_asset_price(self, asset):
        chainlink_price = await self._get_chainlink_price(asset)
        dex_price = await self._get_dex_price(asset)

```

```

    if chainlink_price is None and dex_price is None:
        raise ValueError(f"No price available for {asset}")

```

```

    if chainlink_price is None:
        return dex_price

```

```

    if dex_price is None:
        return chainlink_price

```

```

    deviation = abs(chainlink_price - dex_price) / chainlink_price *
100

```

```

if deviation > self.threshold:
    raise ValueError(f'Price deviation too high: {deviation}%')

return chainlink_price

async def get_collateral_value(self, user_address, assets):
    total_value = 0

    for asset, amount in assets.items():
        price = await self.get_asset_price(asset)
        decimals = await self._get_decimals(asset)

        value = (amount * price) / (10 ** decimals)
        total_value += value

    return total_value

async def check_liquidation(self, user_address, collateral, debt,
liquidation_threshold):
    collateral_value = await self.get_collateral_value(user_address,
collateral)
    debt_value = await self.get_collateral_value(user_address, debt)

    health_factor = (collateral_value * liquidation_threshold) /
debt_value

    return {
        "collateral_value": collateral_value,
        "debt_value": debt_value,
        "health_factor": health_factor,
        "liquidatable": health_factor < 1
    }

async def _get_chainlink_price(self, asset):
    pass

async def _get_dex_price(self, asset):
    pass

async def _get_decimals(self, asset):

```

pass

CHAPTER SUMMARY

This chapter covered price discovery and oracles:

Price discovery in AMMs happens through trading and arbitrage. Every swap moves prices. Arbitrageurs align DEX prices with external markets.

TWAP (Time-Weighted Average Price) resists manipulation by averaging prices over time. Attackers must sustain manipulation across multiple blocks.

Chainlink oracles provide decentralized price feeds from multiple independent nodes. Staleness checks prevent using outdated prices.

On-chain price calculation directly from DEX reserves provides real-time prices but is vulnerable to manipulation.

Manipulation detection compares current prices to historical averages and cross-references multiple sources.

Secure oracles use multiple sources, deviation thresholds, and fallback mechanisms.

Price impact estimation helps users understand the cost of large trades before execution.

Oracle integration patterns for lending protocols combine multiple sources with deviation checks.

Next chapter: Flash Loans and Arbitrage. We explore instant loans and profit strategies.

CHAPTER 5: FLASH LOANS AND ARBITRAGE

Flash loans changed DeFi forever. Borrow millions instantly. No collateral required. Just repay within the same transaction. This chapter covers flash loan mechanics, arbitrage strategies, and

building profitable bots.

SECTION 1: FLASH LOAN FUNDAMENTALS

Traditional loans require collateral. Flash loans require only atomicity - repay in the same transaction or the entire transaction reverts.

How Flash Loans Work:

Transaction begins

- > Borrow 1,000,000 USDC from Aave
- > Execute profitable operations
- > Repay 1,000,000 USDC + fee

Transaction ends

If repayment fails, everything reverts. The lender never loses funds.

Flash Loan Interface:

```
from abc import ABC, abstractmethod
```

```
class IFlashLoanReceiver(ABC):
```

```
    @abstractmethod
    async def execute_operation(
        self,
        assets,
        amounts,
        premiums,
        initiator,
        params
    ):
        pass
```

```
class FlashLoanProvider:
```

```
    def __init__(self, reserves):
```



```
self.reserves = reserves
self.fee_percent = 0.09
```

```
async def flash_loan(self, receiver, assets, amounts, params):
    for asset, amount in zip(assets, amounts):
        if self.reserves.get(asset, 0) < amount:
            raise ValueError(f'Insufficient liquidity for {asset}')
```

```
    premiums = [int(amount * self.fee_percent / 100) for amount in
amounts]
```

```
    for asset, amount in zip(assets, amounts):
        self.reserves[asset] -= amount
```

```
    await receiver.execute_operation(
        assets,
        amounts,
        premiums,
        self,
        params
    )
```

```
    for asset, amount, premium in zip(assets, amounts, premiums):
        expected_return = amount + premium
```

```
    actual_balance = self.reserves.get(asset, 0)
    self.reserves[asset] = actual_balance + expected_return
```

```
    if self.reserves[asset] < expected_return:
        raise ValueError("Flash loan not repaid")
```

SECTION 2: AAVE FLASH LOANS

Aave is the largest flash loan provider. Understanding its interface is essential.

Aave Flash Loan Consumer:

```
class AaveFlashLoanExecutor:
```

```

LENDING_POOL_ABI = [
{
"name": "flashLoan",
"type": "function",
"inputs": [
{"name": "receiverAddress", "type": "address"},
{"name": "assets", "type": "address[]"},
{"name": "amounts", "type": "uint256[]"},
{"name": "modes", "type": "uint256[]"},
{"name": "onBehalfOf", "type": "address"},
{"name": "params", "type": "bytes"},
{"name": "referralCode", "type": "uint16"}
],
"outputs": []
}
]

```

```

def __init__(self, web3, lending_pool_address, receiver_contract):
self.web3 = web3
self.lending_pool = web3.eth.contract(
address=lending_pool_address,
abi=self.LENDING_POOL_ABI
)
self.receiver = receiver_contract

```

```

async def execute_flash_loan(self, assets, amounts, params, wallet):
modes = [0] * len(assets)

```

```

tx = self.lending_pool.functions.flashLoan(
self.receiver,
assets,
amounts,
modes,
wallet.address,
params,
0
).build_transaction({
"from": wallet.address,
"gas": 1000000,

```

```

"gasPrice": self.web3.eth.gas_price,
"nonce": self.web3.eth.get_transaction_count(wallet.address)
})

signed = wallet.sign_transaction(tx)
tx_hash =
self.web3.eth.send_raw_transaction(signed.rawTransaction)

return tx_hash.hex()

```

Flash Loan Receiver Contract (Solidity-like pseudocode):

```

flash_loan_receiver_logic = ""

contract FlashLoanReceiver is IFlashLoanReceiver {

    address public owner;
    address public lendingPool;

    constructor(address _lendingPool) {
        owner = msg.sender;
        lendingPool = _lendingPool;
    }

    function executeOperation(
        address[] calldata assets,
        uint256[] calldata amounts,
        uint256[] calldata premiums,
        address initiator,
        bytes calldata params
    ) external returns (bool) {

        require(msg.sender == lendingPool, "Invalid caller");
        require(initiator == owner, "Invalid initiator");

        (uint8 operation, bytes memory operationParams) =
        abi.decode(params, (uint8, bytes));

        if (operation == 1) {

```

```

executeArbitrage/assets, amounts, operationParams);
} else if (operation == 2) {
executeLiquidation/assets, amounts, operationParams);
}

for (uint i = 0; i < assets.length; i++) {
uint amountOwed = amounts[i] + premiums[i];
IERC20/assets[i]).approve(lendingPool, amountOwed);
}

return true;
}

function executeArbitrage(
address[] memory assets,
uint256[] memory amounts,
bytes memory params
) internal {
// Decode arbitrage parameters
// Execute swaps
// Ensure profit covers premium
}

}

""""

```

SECTION 3: ARBITRAGE FUNDAMENTALS

Arbitrage exploits price differences across markets. Buy low on one exchange. Sell high on another.

Simple Arbitrage Detector:

```

class ArbitrageDetector:

def __init__(self, dex_clients):
self.dexes = dex_clients

```

```
async def find_opportunities(self, token_a, token_b, amount):  
    prices = {}
```

```
    for dex_name, dex in self.dexes.items():  
        try:  
            quote = await dex.get_quote(token_a, token_b, amount)  
            prices[dex_name] = {  
                "amount_out": quote,  
                "price": quote / amount  
            }  
        except Exception as e:  
            continue
```

```
    if len(prices) < 2:  
        return None
```

```
    best_buy = min(prices.items(), key=lambda x: x[1]["price"])  
    best_sell = max(prices.items(), key=lambda x: x[1]["price"])
```

```
    if best_buy[0] == best_sell[0]:  
        return None
```

```
    price_diff = (best_sell[1]["price"] - best_buy[1]["price"]) /  
    best_buy[1]["price"]
```

```
    if price_diff < 0.003:  
        return None
```

```
    return {  
        "buy_dex": best_buy[0],  
        "sell_dex": best_sell[0],  
        "buy_price": best_buy[1]["price"],  
        "sell_price": best_sell[1]["price"],  
        "price_difference_percent": price_diff * 100,  
        "estimated_profit": amount * price_diff  
    }
```

```
    async def calculate_optimal_amount(self, token_a, token_b, buy_dex,  
    sell_dex, max_amount):  
        best_profit = 0
```

```
best_amount = 0
```

```
test_amounts = [max_amount * i / 10 for i in range(1, 11)]
```

```
for amount in test_amounts:
```

```
    buy_quote = await self.dexes[buy_dex].get_quote(token_a, token_b,  
    amount)
```

```
    sell_quote = await self.dexes[sell_dex].get_quote(token_b, token_a,  
    buy_quote)
```

```
    profit = sell_quote - amount
```

```
    gas_cost = 0.01 * amount
```

```
    net_profit = profit - gas_cost
```

```
    if net_profit > best_profit:
```

```
        best_profit = net_profit
```

```
        best_amount = amount
```

```
return best_amount, best_profit
```

Triangular Arbitrage:

```
class TriangularArbitrage:
```

```
    def __init__(self, dex_client):
```

```
        self.dex = dex_client
```

```
    async def find_triangular_opportunity(self, token_a, token_b,  
    token_c, start_amount):
```

```
        try:
```

```
            amount_b = await self.dex.get_quote(token_a, token_b,  
            start_amount)
```

```
            amount_c = await self.dex.get_quote(token_b, token_c, amount_b)
```

```
            final_amount = await self.dex.get_quote(token_c, token_a,  
            amount_c)
```

```
profit = final_amount - start_amount
profit_percent = (profit / start_amount) * 100
```

```
return {
    "path": [token_a, token_b, token_c, token_a],
    "start_amount": start_amount,
    "final_amount": final_amount,
    "profit": profit,
    "profit_percent": profit_percent,
    "profitable": profit_percent > 0.5
}
```

```
except Exception as e:
    return None
```

```
async def find_all_triangular_opportunities(self, tokens, start_token,
start_amount):
    opportunities = []
```

```
    for token_b in tokens:
        if token_b == start_token:
            continue
```

```
    for token_c in tokens:
        if token_c == start_token or token_c == token_b:
            continue
```

```
    opp = await self.find_triangular_opportunity(
        start_token, token_b, token_c, start_amount
    )
```

```
    if opp and opp["profitable"]:
        opportunities.append(opp)
```

```
    return sorted(opportunities, key=lambda x: x["profit"],
reverse=True)
```

SECTION 4: FLASH LOAN ARBITRAGE

Combine flash loans with arbitrage for risk-free profits (minus gas).

Flash Arbitrage Executor:

```
class FlashArbitrageExecutor:
```

```
    def __init__(self, web3, flash_loan_provider, dex_router):
        self.web3 = web3
        self.flash_provider = flash_loan_provider
        self.router = dex_router
```

```
    async def execute_arbitrage(self, opportunity, wallet):
        loan_amount = opportunity["optimal_amount"]
        asset = opportunity["token_a"]
```

```
        buy_path = opportunity["buy_path"]
        sell_path = opportunity["sell_path"]
        buy_dex = opportunity["buy_dex"]
        sell_dex = opportunity["sell_dex"]
```

```
        params = self.encode_arbitrage_params(
            buy_dex, sell_dex, buy_path, sell_path
        )
```

```
        tx_hash = await self.flash_provider.execute_flash_loan(
            [asset],
            [loan_amount],
            params,
            wallet
        )
```

```
        return tx_hash
```

```
    def encode_arbitrage_params(self, buy_dex, sell_dex, buy_path,
                                sell_path):
        import eth_abi
```

```
        return eth_abi.encode(
            ["uint8", "address", "address", "address[]", "address[]"],
            [1, buy_dex, sell_dex, buy_path, sell_path]
```


)

```
async def simulate_arbitrage(self, opportunity):  
    loan_amount = opportunity["optimal_amount"]  
    premium = int(loan_amount * 0.0009)
```

```
    buy_output = await self._simulate_swap(  
        opportunity["buy_dex"],  
        opportunity["buy_path"],  
        loan_amount  
    )
```

```
    sell_output = await self._simulate_swap(  
        opportunity["sell_dex"],  
        opportunity["sell_path"],  
        buy_output  
    )
```

```
    gas_cost = await self._estimate_gas_cost()
```

```
    gross_profit = sell_output - loan_amount  
    net_profit = gross_profit - premium - gas_cost
```

```
    return {  
        "loan_amount": loan_amount,  
        "premium": premium,  
        "gas_cost": gas_cost,  
        "gross_profit": gross_profit,  
        "net_profit": net_profit,  
        "profitable": net_profit > 0  
    }
```

```
async def _simulate_swap(self, dex, path, amount_in):  
    pass
```

```
async def _estimate_gas_cost(self):  
    gas_price = self.web3.eth.gas_price  
    estimated_gas = 500000  
    return gas_price * estimated_gas
```

SECTION 5: MEV AND FRONTRUNNING

Maximal Extractable Value (MEV) is profit extracted by reordering transactions. Frontrunning is placing your transaction before a target.

Frontrunning Detection:

```
class FrontrunningDetector:
```

```
    def __init__(self, web3):
        self.web3 = web3
        self.pending_txs = {}
```

```
    async def monitor_mempool(self):
        pending_filter = self.web3.eth.filter("pending")
```

```
        while True:
            new_entries = pending_filter.get_new_entries()
```

```
            for tx_hash in new_entries:
                try:
                    tx = self.web3.eth.get_transaction(tx_hash)
                    await self.analyze_transaction(tx)
                except Exception:
                    continue
```

```
    async def analyze_transaction(self, tx):
        if not tx.get("input"):
            return
```

```
        method_id = tx["input"][:10]
```

```
        swap_methods = [
            "0x7ff36ab5",
            "0x18cbase5",
            "0x38ed1739",
            "0x8803dbee"
        ]
```

```

if method_id in swap_methods:
    decoded = self.decode_swap(tx["input"])

    self.pending_txs[tx["hash"].hex()] = {
        "tx": tx,
        "decoded": decoded,
        "timestamp": time.time()
    }

    opportunity = self.evaluate_frontrun_opportunity(tx, decoded)

    if opportunity["profitable"]:
        return opportunity

def decode_swap(self, input_data):
    return {}

def evaluate_frontrun_opportunity(self, tx, decoded):
    return {"profitable": False}

```

MEV Protection:

```

class MEVProtection:

    def __init__(self, flashbots_relay):
        self.flashbots = flashbots_relay

    async def submit_private_transaction(self, signed_tx, target_block):
        bundle = {
            "signedTransactions": [signed_tx],
            "blockNumber": hex(target_block)
        }

        response = await self.flashbots.send_bundle(bundle)

        return response

    async def submit_with_backrun_protection(self, tx, wallet):

```

```
current_block = self.web3.eth.block_number
```

```
signed_tx = wallet.sign_transaction(tx)
```

```
for target_block in range(current_block + 1, current_block + 4):  
    await self.submit_private_transaction(  
        signed_tx.rawTransaction.hex(),  
        target_block  
    )
```

```
return signed_tx.hash.hex()
```

```
def calculate_sandwich_risk(self, trade_amount, pool_reserves):  
    impact = trade_amount / pool_reserves
```

```
    if impact > 0.01:  
        return "high"  
    elif impact > 0.005:  
        return "medium"  
    else:  
        return "low"
```

SECTION 6: LIQUIDATION BOTS

Liquidation bots profit by liquidating undercollateralized positions.

Liquidation Bot:

```
class LiquidationBot:
```

```
    def __init__(self, web3, lending_protocol, flash_loan_provider):  
        self.web3 = web3  
        self.protocol = lending_protocol  
        self.flash_provider = flash_loan_provider
```

```
    async def monitor_positions(self):  
        while True:  
            unhealthy_positions = await self.protocol.get_unhealthy_positions()
```

```

for position in unhealthy_positions:
    profit = await self.calculate_liquidation_profit(position)

    if profit > 0:
        await self.execute_liquidation(position)

        await asyncio.sleep(1)

    async def calculate_liquidation_profit(self, position):
        debt_to_cover = position["debt"] * 0.5

        collateral_received = await self.protocol.get_liquidation_bonus(
            position["collateral_asset"],
            debt_to_cover
        )

        collateral_value = await self.get_market_value(
            position["collateral_asset"],
            collateral_received
        )

        flash_loan_fee = debt_to_cover * 0.0009
        gas_cost = await self.estimate_gas_cost()

        profit = collateral_value - debt_to_cover - flash_loan_fee - gas_cost

    return profit

    async def execute_liquidation(self, position):
        debt_to_cover = int(position["debt"] * 0.5)

        params = self.encode_liquidation_params(
            position["user"],
            position["collateral_asset"],
            position["debt_asset"]
        )

        tx_hash = await self.flash_provider.execute_flash_loan(
            [position["debt_asset"]],
            [debt_to_cover],

```

```
params,  
self.wallet  
)
```

```
return tx_hash
```

```
def encode_liquidation_params(self, user, collateral, debt):  
import eth_abi
```

```
return eth_abi.encode(  
["uint8", "address", "address", "address"],  
[2, user, collateral, debt]  
)
```

```
async def get_market_value(self, asset, amount):  
pass
```

```
async def estimate_gas_cost(self):  
pass
```

SECTION 7: ARBITRAGE BOT ARCHITECTURE

Complete architecture for a production arbitrage bot.

Production Arbitrage Bot:

```
class ProductionArbitrageBot:
```

```
def __init__(self, config):  
self.config = config  
self.web3 = Web3(Web3.HTTPProvider(config.rpc_url))  
self.wallet = Account.from_key(config.private_key)
```

```
self.dexes = self._initialize_dexes()  
self.flash_provider = self._initialize_flash_provider()
```

```
self.min_profit_threshold = config.min_profit_eth  
self.max_gas_price = config.max_gas_gwei
```

```
self.opportunities_found = 0
self.opportunities_executed = 0
self.total_profit = 0
```

```
self.running = False
```

```
def _initialize_dexes(self):
    return {
        "uniswap_v2": UniswapV2Client(self.web3,
self.config.uniswap_v2_router),
        "sushiswap": UniswapV2Client(self.web3,
self.config.sushiswap_router),
        "uniswap_v3": UniswapV3Client(self.web3,
self.config.uniswap_v3_router)
    }
```

```
def _initialize_flash_provider(self):
    return AaveFlashLoanExecutor(
self.web3,
self.config.aave_lending_pool,
self.config.flash_receiver
    )
```

```
async def start(self):
    self.running = True
```

```
asyncio.create_task(self.opportunity_scanner())
asyncio.create_task(self.execution_loop())
asyncio.create_task(self.metrics_reporter())
```

```
async def stop(self):
    self.running = False
```

```
async def opportunity_scanner(self):
    detector = ArbitrageDetector(self.dexes)
```

```
while self.running:
    try:
        for pair in self.config.token_pairs:
            token_a, token_b = pair
```

```

for amount in self.config.test_amounts:
    opp = await detector.find_opportunities(
        token_a, token_b, amount
    )

    if opp and await self.is_profitable(opp):
        await self.opportunity_queue.put(opp)
        self.opportunities_found += 1

except Exception as e:
    print(f"Scanner error: {e}")

await asyncio.sleep(0.1)

async def is_profitable(self, opportunity):
    gas_price = self.web3.eth.gas_price

    if gas_price > self.max_gas_price * 10**9:
        return False

    simulation = await self.simulate_execution(opportunity)

    return simulation["net_profit"] > self.min_profit_threshold

async def simulate_execution(self, opportunity):
    executor = FlashArbitrageExecutor(
        self.web3,
        self.flash_provider,
        None
    )

    return await executor.simulate_arbitrage(opportunity)

async def execution_loop(self):
    while self.running:
        try:
            opportunity = await asyncio.wait_for(
                self.opportunity_queue.get(),
                timeout=1.0
            )

```


)

```
if await self.is_profitable(opportunity):  
    result = await self.execute(opportunity)
```

```
if result["success"]:  
    self.opportunities_executed += 1  
    self.total_profit += result["profit"]
```

```
except asyncio.TimeoutError:  
    continue  
except Exception as e:  
    print(f"Execution error: {e}")
```

```
async def execute(self, opportunity):  
    try:  
        executor = FlashArbitrageExecutor(  
            self.web3,  
            self.flash_provider,  
            None  
        )
```

```
        tx_hash = await executor.execute_arbitrage(opportunity,  
            self.wallet)
```

```
        receipt = await self.wait_for_receipt(tx_hash)
```

```
        if receipt["status"] == 1:  
            profit = await self.calculate_actual_profit(receipt)  
            return {"success": True, "tx_hash": tx_hash, "profit": profit}  
        else:  
            return {"success": False, "tx_hash": tx_hash, "error": "Reverted"}
```

```
    except Exception as e:  
        return {"success": False, "error": str(e)}
```

```
    async def wait_for_receipt(self, tx_hash, timeout = 120):  
        start = time.time()
```

```
        while time.time() - start < timeout:
```

```

try:
    receipt = self.web3.eth.get_transaction_receipt(tx_hash)
    if receipt:
        return receipt
    except Exception:
        pass

    await asyncio.sleep(1)

    raise TimeoutError("Transaction not confirmed")

    async def calculate_actual_profit(self, receipt):
        pass

    async def metrics_reporter(self):
        while self.running:
            print(f"Opportunities Found: {self.opportunities_found}")
            print(f"Opportunities Executed: {self.opportunities_executed}")
            print(f"Total Profit: {self.total_profit} ETH")

        await asyncio.sleep(60)

```

SECTION 8: RISK MANAGEMENT

Arbitrage bots face real risks. Gas price spikes. Failed transactions. Smart contract bugs.

Risk Management System:

```

class ArbitrageRiskManager:

    def __init__(self, config):
        self.max_position_size = config.max_position_size
        self.max_gas_price = config.max_gas_price
        self.max_daily_loss = config.max_daily_loss
        self.min_profit_ratio = config.min_profit_ratio

        self.daily_pnl = 0
        self.failed_txs = 0

```

```

self.last_reset = datetime.utcnow().date()

def check_opportunity(self, opportunity):
    self._maybe_reset_daily()

    if self.daily_pnl < -self.max_daily_loss:
        return False, "Daily loss limit reached"

    if opportunity["amount"] > self.max_position_size:
        return False, "Position size too large"

    profit_ratio = opportunity["profit"] / opportunity["amount"]
    if profit_ratio < self.min_profit_ratio:
        return False, "Profit ratio too low"

    return True, None

def check_gas_price(self, current_gas_price):
    if current_gas_price > self.max_gas_price:
        return False, f'Gas price {current_gas_price} exceeds max {self.max_gas_price}'

    return True, None

def record_result(self, success, profit=0):
    if success:
        self.daily_pnl += profit
    else:
        self.failed_txs += 1
        self.daily_pnl -= 0.01

def _maybe_reset_daily(self):
    today = datetime.utcnow().date()

    if today != self.last_reset:
        self.daily_pnl = 0
        self.failed_txs = 0
        self.last_reset = today

def get_status(self):

```

```

return {
    "daily_pnl": self.daily_pnl,
    "failed_txs": self.failed_txs,
    "can_trade": self.daily_pnl > -self.max_daily_loss
}

```

Gas Price Monitor:

```

class GasPriceMonitor:

```

```

    def __init__(self, web3):
        self.web3 = web3
        self.price_history = []
        self.max_history = 100

```

```

    async def get_current_gas_price(self):
        return self.web3.eth.gas_price

```

```

    async def get_optimal_gas_price(self, urgency="normal"):
        current = await self.get_current_gas_price()

```

```

        multipliers = {
            "slow": 0.9,
            "normal": 1.0,
            "fast": 1.2,
            "instant": 1.5
        }

```

```

        return int(current * multipliers.get(urgency, 1.0))

```

```

    async def predict_gas_price(self, blocks_ahead):
        if len(self.price_history) < 10:
            return await self.get_current_gas_price()

```

```

        recent_prices = [p["price"] for p in self.price_history[-10:]]
        avg_price = sum(recent_prices) / len(recent_prices)

```

```

        trend = (recent_prices[-1] - recent_prices[0]) / recent_prices[0]

```

```
predicted = avg_price * (1 + trend * blocks_ahead / 10)
```

```
return int(predicted)
```

```
async def update_history(self):
```

```
    current = await self.get_current_gas_price()
```

```
    self.price_history.append({
```

```
        "price": current,
```

```
        "timestamp": time.time()
```

```
    })
```

```
    if len(self.price_history) > self.max_history:
```

```
        self.price_history.pop(0)
```

CHAPTER SUMMARY

This chapter covered flash loans and arbitrage:

Flash loans provide instant, uncollateralized borrowing that must be repaid within a single transaction.

Aave is the primary flash loan provider with a standardized receiver interface for executing operations.

Arbitrage exploits price differences across exchanges. Simple arbitrage trades between two DEXes. Triangular arbitrage uses three-token paths.

Flash loan arbitrage combines instant borrowing with arbitrage for capital-efficient profits.

MEV (Maximal Extractable Value) represents profit from transaction ordering. Frontrunning places transactions ahead of targets.

Liquidation bots profit by liquidating undercollateralized positions in lending protocols.

Production arbitrage bots require robust architecture: opportunity

scanning, simulation, execution, and metrics.

Risk management protects against losses: position limits, gas price monitoring, daily loss limits.

Next chapter: Advanced Swap Routing. We optimize trade execution across multiple paths.

CHAPTER 6: ADVANCED SWAP ROUTING

Large trades suffer from price impact. Smart routing splits trades across multiple pools and paths. This chapter covers pathfinding algorithms, split routing, and building DEX aggregators that find optimal execution.

SECTION 1: THE ROUTING PROBLEM

A simple swap might have dozens of possible paths. ETH to USDC could go direct, through WBTC, through DAI, or split across multiple routes. Finding the best execution is a complex optimization problem.

Route Discovery:

```
class RouteFinder:
```

```
    def __init__(self, graph):
```

```
        self.graph = graph
```

```
        self.max_hops = 4
```

```
    def find_all_paths(self, source, target, max_depth = None):
```

```
        if max_depth is None:
```

```
            max_depth = self.max_hops
```

```
        paths = []
```

```
        visited = set()
```

```
    def dfs(current, path):
```

```
        if len(path) > max_depth:
```

```
            return
```

```
if current == target:
    paths.append(path.copy())
    return
```

```
visited.add(current)
```

```
for neighbor in self.graph.get_neighbors(current):
    if neighbor not in visited:
        path.append(neighbor)
        dfs(neighbor, path)
        path.pop()
```

```
visited.remove(current)
```

```
dfs(source, [source])
```

```
return paths
```

```
def find_paths_with_liquidity(self, source, target, min_liquidity):
    all_paths = self.find_all_paths(source, target)
```

```
    valid_paths = []
```

```
    for path in all_paths:
        path_liquidity = self.get_path_liquidity(path)
```

```
    if path_liquidity >= min_liquidity:
        valid_paths.append({
            "path": path,
            "liquidity": path_liquidity
        })
```

```
    return sorted(valid_paths, key=lambda x: x["liquidity"],
reverse=True)
```

```
def get_path_liquidity(self, path):
    min_liquidity = float("inf")
```

```
    for i in range(len(path) - 1):
```

```
token_a = path[i]
token_b = path[i + 1]

pair_liquidity = self.graph.get_liquidity(token_a, token_b)
min_liquidity = min(min_liquidity, pair_liquidity)

return min_liquidity
```

Liquidity Graph:

```
class LiquidityGraph:
```

```
    def __init__(self):
        self.edges = {}
        self.pools = {}
```

```
    def add_pool(self, pool_address, token_a, token_b, reserve_a,
reserve_b, fee):
        self.pools[pool_address] = {
            "token_a": token_a,
            "token_b": token_b,
            "reserve_a": reserve_a,
            "reserve_b": reserve_b,
            "fee": fee
        }
```

```
        if token_a not in self.edges:
            self.edges[token_a] = {}
        if token_b not in self.edges:
            self.edges[token_b] = {}
```

```
        if token_b not in self.edges[token_a]:
            self.edges[token_a][token_b] = []
        if token_a not in self.edges[token_b]:
            self.edges[token_b][token_a] = []
```

```
        self.edges[token_a][token_b].append(pool_address)
        self.edges[token_b][token_a].append(pool_address)
```



```

def get_neighbors(self, token):
    return list(self.edges.get(token, {}).keys())

def get_pools_for_pair(self, token_a, token_b):
    return self.edges.get(token_a, {}).get(token_b, [])

def get_liquidity(self, token_a, token_b):
    pools = self.get_pools_for_pair(token_a, token_b)

    total_liquidity = 0

    for pool_address in pools:
        pool = self.pools[pool_address]
        if pool["token_a"] == token_a:
            total_liquidity += pool["reserve_a"]
        else:
            total_liquidity += pool["reserve_b"]

    return total_liquidity

```

SECTION 2: PATH EVALUATION

Once paths are found, evaluate which produces the best output.

Path Evaluator:

```

class PathEvaluator:

    def __init__(self, graph):
        self.graph = graph

    def evaluate_path(self, path, amount_in):
        current_amount = amount_in

        for i in range(len(path) - 1):
            token_in = path[i]
            token_out = path[i + 1]

            pools = self.graph.get_pools_for_pair(token_in, token_out)

```

```
if not pools:  
    return 0
```

```
best_output = 0
```

```
for pool_address in pools:  
    pool = self.graph.pools[pool_address]  
    output = self.calculate_output(pool, token_in, current_amount)  
    best_output = max(best_output, output)
```

```
current_amount = best_output
```

```
if current_amount == 0:  
    return 0
```

```
return current_amount
```

```
def calculate_output(self, pool, token_in, amount_in):  
    if pool["token_a"] == token_in:  
        reserve_in = pool["reserve_a"]  
        reserve_out = pool["reserve_b"]  
    else:  
        reserve_in = pool["reserve_b"]  
        reserve_out = pool["reserve_a"]
```

```
fee_multiplier = 10000 - int(pool["fee"] * 100)
```

```
amount_in_with_fee = amount_in * fee_multiplier  
numerator = amount_in_with_fee * reserve_out  
denominator = reserve_in * 10000 + amount_in_with_fee
```

```
return numerator // denominator
```

```
def find_best_path(self, paths, amount_in):  
    best_path = None  
    best_output = 0
```

```
for path in paths:  
    output = self.evaluate_path(path, amount_in)
```

```

if output > best_output:
    best_output = output
    best_path = path

return best_path, best_output

def calculate_price_impact(self, path, amount_in):
    small_amount = amount_in // 1000
    small_output = self.evaluate_path(path, small_amount)

    if small_output == 0:
        return float("inf")

    spot_price = small_output / small_amount

    full_output = self.evaluate_path(path, amount_in)
    execution_price = full_output / amount_in

    price_impact = (spot_price - execution_price) / spot_price * 100

    return price_impact

```

SECTION 3: SPLIT ROUTING

Large trades benefit from splitting across multiple paths.

Split Router:

```

class SplitRouter:

    def __init__(self, graph, evaluator):
        self.graph = graph
        self.evaluator = evaluator

    def find_optimal_split(self, source, target, amount_in,
                           max_splits=4):
        finder = RouteFinder(self.graph)
        all_paths = finder.find_all_paths(source, target)

```

```

if len(all_paths) == 0:
    return None

if len(all_paths) == 1:
    output = self.evaluator.evaluate_path(all_paths[0], amount_in)
    return [{
        "path": all_paths[0],
        "amount": amount_in,
        "output": output,
        "percentage": 100
    }]

ranked_paths = []
for path in all_paths:
    output = self.evaluator.evaluate_path(path, amount_in)
    ranked_paths.append({"path": path, "output": output})

ranked_paths.sort(key=lambda x: x["output"], reverse=True)
top_paths = ranked_paths[:max_splits]

best_split = self._optimize_split(top_paths, amount_in)

return best_split

def _optimize_split(self, paths, total_amount, steps=10):
    n_paths = len(paths)

    if n_paths == 1:
        output = self.evaluator.evaluate_path(paths[0]["path"],
        total_amount)
        return [{
            "path": paths[0]["path"],
            "amount": total_amount,
            "output": output,
            "percentage": 100
        }]

    best_output = 0
    best_allocation = None

```

```
for allocation in self._generate_allocations(n_paths, steps):
    amounts = [int(total_amount * a / steps) for a in allocation]
```

```
remainder = total_amount - sum(amounts)
amounts[0] += remainder
```

```
total_output = 0
for i, path_info in enumerate(paths):
    if amounts[i] > 0:
        output = self.evaluator.evaluate_path(path_info["path"],
        amounts[i])
        total_output += output
```

```
if total_output > best_output:
    best_output = total_output
    best_allocation = amounts
```

```
result = []
for i, path_info in enumerate(paths):
    if best_allocation[i] > 0:
        output = self.evaluator.evaluate_path(path_info["path"],
        best_allocation[i])
        result.append({
            "path": path_info["path"],
            "amount": best_allocation[i],
            "output": output,
            "percentage": best_allocation[i] * 100 / total_amount
        })
```

```
return result
```

```
def _generate_allocations(self, n, total):
    if n == 1:
        yield [total]
    return
```

```
for i in range(total + 1):
    for rest in self._generate_allocations(n - 1, total - i):
        yield [i] + rest
```

SECTION 4: MULTI-DEX ROUTING

Route across multiple DEXes for better execution.

Multi-DEX Router:

```
class MultiDEXRouter:
```

```
    def __init__(self, dex_configs):
        self.dexes = {}
```

```
    for name, config in dex_configs.items():
        self.dexes[name] = {
            "router": config["router_address"],
            "factory": config["factory_address"],
            "fee": config["fee"],
            "graph": LiquidityGraph()
        }
```

```
    async def load_pools(self, token_list):
        for dex_name, dex_config in self.dexes.items():
            for i, token_a in enumerate(token_list):
                for token_b in token_list[i + 1:]:
                    pool_info = await self._get_pool_info(
                        dex_config["factory"],
                        token_a,
                        token_b
                    )
```

```
                    if pool_info:
                        dex_config["graph"].add_pool(
                            pool_info["address"],
                            token_a,
                            token_b,
                            pool_info["reserve_a"],
                            pool_info["reserve_b"],
                            dex_config["fee"]
                        )
```

```

def find_best_route_across_dexes(self, source, target, amount_in):
    all_routes = []

    for dex_name, dex_config in self.dexes.items():
        finder = RouteFinder(dex_config["graph"])
        evaluator = PathEvaluator(dex_config["graph"])

        paths = finder.find_all_paths(source, target)

        for path in paths:
            output = evaluator.evaluate_path(path, amount_in)

            all_routes.append({
                "dex": dex_name,
                "path": path,
                "output": output,
                "router": dex_config["router"]
            })

        all_routes.sort(key = lambda x: x["output"], reverse = True)

    return all_routes[:10]

def find_split_route_across_dexes(self, source, target, amount_in):
    single_routes = self.find_best_route_across_dexes(source, target,
amount_in)

    if len(single_routes) < 2:
        return single_routes[:1]

    top_routes = single_routes[:4]

    best_output = 0
    best_split = None

    for allocation in self.generate_splits(len(top_routes), 10):
        amounts = [int(amount_in * a / 10) for a in allocation]

        remainder = amount_in - sum(amounts)

```

```

if remainder > 0:
    amounts[0] += remainder

total_output = 0
for i, route in enumerate(top_routes):
    if amounts[i] > 0:
        evaluator = PathEvaluator(self.dexes[route["dex"]]["graph"])
        output = evaluator.evaluate_path(route["path"], amounts[i])
        total_output += output

if total_output > best_output:
    best_output = total_output
    best_split = [
        {**route, "amount": amounts[i], "output_share": 0}
        for i, route in enumerate(top_routes)
        if amounts[i] > 0
    ]

return best_split

def _generate_splits(self, n, total):
    if n == 1:
        yield [total]
    return

for i in range(total + 1):
    for rest in self._generate_splits(n - 1, total - i):
        yield [i] + rest

async def _get_pool_info(self, factory, token_a, token_b):
    pass

```

SECTION 5: ROUTE CACHING

Computing routes is expensive. Cache frequently used routes.

Route Cache:

```
class RouteCache:
```



```

def __init__(self, ttl_seconds=60):
    self.cache = {}
    self.ttl = ttl_seconds

def get_cache_key(self, source, target, amount_in):
    amount_bucket = amount_in // (10 ** 16)
    return f"{source}:{target}:{amount_bucket}"

def get(self, source, target, amount_in):
    key = self.get_cache_key(source, target, amount_in)

    if key in self.cache:
        cached = self.cache[key]

        if time.time() - cached["timestamp"] < self.ttl:
            return cached["routes"]

    del self.cache[key]

    return None

def set(self, source, target, amount_in, routes):
    key = self.get_cache_key(source, target, amount_in)

    self.cache[key] = {
        "routes": routes,
        "timestamp": time.time()
    }

def invalidate(self, token=None):
    if token is None:
        self.cache.clear()
    return

keys_to_delete = [
    key for key in self.cache.keys()
    if token in key
]

```

```
for key in keys_to_delete:
    del self.cache[key]
```

Cached Router:

```
class CachedRouter:
```

```
    def __init__(self, router, cache):
        self.router = router
        self.cache = cache
```

```
    async def find_best_route(self, source, target, amount_in):
        cached = self.cache.get(source, target, amount_in)
```

```
        if cached:
            return cached
```

```
        routes = self.router.find_best_route_across_dexes(source, target,
                                                            amount_in)
```

```
        self.cache.set(source, target, amount_in, routes)
```

```
        return routes
```

SECTION 6: REAL-TIME ROUTE UPDATES

Pool reserves change with every trade. Keep routes updated.

Real-Time Route Manager:

```
class RealTimeRouteManager:
```

```
    def __init__(self, web3, router, pools_to_watch):
        self.web3 = web3
        self.router = router
        self.pools = pools_to_watch
        self.running = False
```

```
async def start(self):
    self.running = True
```

```
asyncio.create_task(self.poll_reserves())
asyncio.create_task(self.listen_to_swaps())
```

```
async def stop(self):
    self.running = False
```

```
async def poll_reserves(self):
    while self.running:
        for pool_address in self.pools:
            try:
                reserves = await self._get_reserves(pool_address)
                await self.update_pool(pool_address, reserves)
            except Exception as e:
                pass
```

```
await asyncio.sleep(5)
```

```
async def listen_to_swaps(self):
    swap_topic =
    self.web3.keccak(text="Swap(address,uint256,uint256,uint256,uint256,ad
```

```
filter_params = {
    "topics": [swap_topic.hex()],
    "address": self.pools
}
```

```
swap_filter = self.web3.eth.filter(filter_params)
```

```
while self.running:
    try:
        new_events = swap_filter.get_new_entries()
```

```
    for event in new_events:
        pool_address = event["address"]
        reserves = await self._get_reserves(pool_address)
        await self.update_pool(pool_address, reserves)
```

```
except Exception as e:  
    pass
```

```
await asyncio.sleep(1)
```

```
async def update_pool(self, pool_address, reserves):  
    for dex_name, dex_config in self.router.dexes.items():  
        if pool_address in dex_config["graph"].pools:  
            pool = dex_config["graph"].pools[pool_address]  
            pool["reserve_a"] = reserves[0]  
            pool["reserve_b"] = reserves[1]
```

```
async def _get_reserves(self, pool_address):  
    pass
```

SECTION 7: EXECUTION OPTIMIZATION

Execute routes efficiently to minimize gas and slippage.

Execution Optimizer:

```
class ExecutionOptimizer:  
  
    def __init__(self, web3):  
        self.web3 = web3  
  
    def build_multicall(self, routes):  
        calls = []  
  
        for route in routes:  
            if route["amount"] == 0:  
                continue  
  
            call_data = self.encode_swap(route)  
  
            calls.append({  
                "target": route["router"],  
                "callData": call_data,  
                "value": route.get("value", 0)
```

```
})
```

```
return calls
```

```
def encode_swap(self, route):  
    if len(route["path"]) == 2:  
        return self._encode_simple_swap(route)  
    else:  
        return self._encode_multi_hop_swap(route)
```

```
def _encode_simple_swap(self, route):  
    pass
```

```
def _encode_multi_hop_swap(self, route):  
    pass
```

```
async def execute_with_simulation(self, routes, wallet):  
    calls = self.build_multicall(routes)
```

```
    try:  
        result = await self._simulate_calls(calls)
```

```
    if not result["success"]:  
        raise ValueError(f'Simulation failed: {result["error"]}'))
```

```
    except Exception as e:  
        raise ValueError(f'Simulation error: {e}')
```

```
    tx = await self._build_multicall_tx(calls, wallet)
```

```
    signed = wallet.sign_transaction(tx)  
    tx_hash =  
    self.web3.eth.send_raw_transaction(signed.rawTransaction)
```

```
    return tx_hash.hex()
```

```
async def _simulate_calls(self, calls):  
    pass
```

```
async def _build_multicall_tx(self, calls, wallet):
```

pass

Gas Estimation:

```
class GasEstimator:
```

```
    BASE_GAS = 21000
```

```
    SWAP_GAS = 150000
```

```
    APPROVAL_GAS = 50000
```

```
    MULTICALL_OVERHEAD = 30000
```

```
    def estimate_route_gas(self, routes):
```

```
        total_gas = self.BASE_GAS
```

```
        for route in routes:
```

```
            hops = len(route["path"]) - 1
```

```
            total_gas += self.SWAP_GAS * hops
```

```
        if len(routes) > 1:
```

```
            total_gas += self.MULTICALL_OVERHEAD
```

```
        return int(total_gas * 1.2)
```

```
    def estimate_total_cost(self, routes, gas_price):
```

```
        gas = self.estimate_route_gas(routes)
```

```
        return gas * gas_price
```

```
    def is_profitable_after_gas(self, routes, expected_output,  
                                input_amount, gas_price, token_price):
```

```
        gas_cost_wei = self.estimate_total_cost(routes, gas_price)
```

```
        gas_cost_tokens = gas_cost_wei * token_price / 10**18
```

```
        net_output = expected_output - gas_cost_tokens
```

```
        return net_output > input_amount
```

SECTION 8: COMPLETE ROUTING ENGINE

Putting all components together.

Complete Routing Engine:

```
class RoutingEngine:
```

```
    def __init__(self, config, web3):
        self.web3 = web3
        self.config = config
```

```

        self.dex_configs = {
            "uniswap_v2": {
                "router_address": config.uniswap_v2_router,
                "factory_address": config.uniswap_v2_factory,
                "fee": 0.3
            },
            "sushiswap": {
                "router_address": config.sushiswap_router,
                "factory_address": config.sushiswap_factory,
                "fee": 0.3
            }
        }
```

```
        self.router = MultiDEXRouter(self.dex_configs)
        self.cache = RouteCache(ttl_seconds=30)
        self.cached_router = CachedRouter(self.router, self.cache)
        self.optimizer = ExecutionOptimizer(web3)
        self.gas_estimator = GasEstimator()
```

```

        async def initialize(self, token_list):
            await self.router.load_pools(token_list)
```

```

        async def get_quote(self, source, target, amount_in):
            routes = await self.cached_router.find_best_route(source, target,
            amount_in)
```

```

            if not routes:
                return None
```

```

            best_route = routes[0]
```

```
split_routes = self.router.find_split_route_across_dexes(source,
target, amount_in)
```

```
single_output = best_route["output"]
split_output = sum(r["output"] for r in split_routes) if split_routes
else 0
```

```
if split_output > single_output * 1.005:
```

```
    return {
        "routes": split_routes,
        "total_output": split_output,
        "split": True
    }
```

```
else:
```

```
    return {
        "routes": [best_route],
        "total_output": single_output,
        "split": False
    }
```

```
async def execute_swap(self, source, target, amount_in, min_output,
wallet):
```

```
    quote = await self.get_quote(source, target, amount_in)
```

```
    if not quote:
```

```
        raise ValueError("No route found")
```

```
    if quote["total_output"] < min_output:
```

```
        raise ValueError(f'Output {quote["total_output"]} below minimum
{min_output}')
```

```
    gas_price = self.web3.eth.gas_price
```

```
    if not self.gas_estimator.is_profitable_after_gas(
```

```
        quote["routes"],
```

```
        quote["total_output"],
```

```
        amount_in,
```

```
        gas_price,
```

```
        1
```

```
    ):
```



```
raise ValueError("Gas cost makes trade unprofitable")
```

```
tx_hash = await self.optimizer.execute_with_simulation(  
    quote["routes"],  
    wallet  
)
```

```
return {  
    "tx_hash": tx_hash,  
    "expected_output": quote["total_output"],  
    "routes": quote["routes"]  
}
```

```
async def get_all_quotes(self, source, target, amount_in):  
    quotes = []
```

```
    routes = self.router.find_best_route_across_dexes(source, target,  
    amount_in)
```

```
    for route in routes[:5]:  
        evaluator = PathEvaluator(self.router.dexes[route["dex"]]  
        ["graph"])  
        price_impact = evaluator.calculate_price_impact(route["path"],  
        amount_in)
```

```
        gas = self.gas_estimator.estimate_route_gas([route])
```

```
        quotes.append({  
            "dex": route["dex"],  
            "path": route["path"],  
            "output": route["output"],  
            "price_impact": price_impact,  
            "estimated_gas": gas  
        })
```

```
    return quotes
```

CHAPTER SUMMARY

This chapter covered advanced swap routing:

Route discovery uses graph algorithms to find all possible paths between tokens.

Path evaluation calculates expected output considering reserves, fees, and price impact.

Split routing divides large trades across multiple paths to reduce overall price impact.

Multi-DEX routing searches across different DEXes to find optimal execution.

Route caching improves performance by storing frequently-used routes with time-based invalidation.

Real-time updates keep route data current by monitoring pool reserves and swap events.

Execution optimization builds efficient multicall transactions and validates through simulation.

Gas estimation ensures trades remain profitable after transaction costs.

Next chapter: DEX Aggregation. We build complete aggregator systems like 1inch.

CHAPTER 7: DEX AGGREGATION

DEX aggregators find the best prices across all exchanges. 1inch, Paraswap, 0x - they search hundreds of liquidity sources to optimize execution. This chapter builds a complete aggregator system from scratch.

SECTION 1: AGGREGATOR ARCHITECTURE

Aggregators are middleware between users and DEXes. They find optimal routes, split trades, and execute across multiple protocols.

Core Components:

Liquidity Source Registry: Tracks all available DEXes and pools

Price Discovery Engine: Fetches quotes from all sources

Routing Algorithm: Finds optimal execution path

Execution Layer: Builds and executes transactions

Gas Optimization: Minimizes transaction costs

Aggregator Core:

class DEXAggregator:

```
def __init__(self, config, web3):
```

```
    self.web3 = web3
```

```
    self.config = config
```

```
    self.sources = LiquiditySourceRegistry()
```

```
    self.price_engine = PriceDiscoveryEngine(self.sources)
```

```
    self.router = AggregatorRouter(self.sources)
```

```
    self.executor = AggregatorExecutor(web3)
```

```
    self.gas_optimizer = GasOptimizer(web3)
```

```
    async def initialize(self):
```

```
        await self.sources.load_all_sources()
```

```
    async def get_quote(self, params):
```

```
        token_in = params["token_in"]
```

```
        token_out = params["token_out"]
```

```
        amount_in = params["amount_in"]
```

```
        slippage = params.get("slippage", 0.5)
```

```
        all_quotes = await self.price_engine.get_all_quotes(
```

```
            token_in, token_out, amount_in
```

```
        )
```

```
        optimal_route = await self.router.find_optimal_route(
```

```
            token_in, token_out, amount_in, all_quotes
```

```
        )
```

```

gas_estimate = await
self.gas_optimizer.estimate_gas(optimal_route)

min_output = int(optimal_route["expected_output"] * (100 -
slippage) / 100)

return {
    "token_in": token_in,
    "token_out": token_out,
    "amount_in": amount_in,
    "expected_output": optimal_route["expected_output"],
    "min_output": min_output,
    "route": optimal_route["route"],
    "sources": optimal_route["sources"],
    "gas_estimate": gas_estimate,
    "price_impact": optimal_route["price_impact"]
}

async def execute_swap(self, quote, wallet):
    tx = await self.executor.build_transaction(quote, wallet)

    signed = wallet.sign_transaction(tx)
    tx_hash =
self.web3.eth.send_raw_transaction(signed.rawTransaction)

    return tx_hash.hex()

```

SECTION 2: LIQUIDITY SOURCE REGISTRY

Track all DEXes, pools, and their characteristics.

Liquidity Source Registry:

```

class LiquiditySourceRegistry:

    def __init__(self):
        self.sources = {}
        self.pools = {}
        self.token_pairs = {}

```

```
def register_source(self, source_id, source_config):
    self.sources[source_id] = {
        "name": source_config["name"],
        "type": source_config["type"],
        "router": source_config["router_address"],
        "factory": source_config.get("factory_address"),
        "fee_tiers": source_config.get("fee_tiers", [0.3]),
        "adapter": source_config["adapter_class"]
    }
```

```
async def load_all_sources(self):
    uniswap_v2_adapter = UniswapV2Adapter(
        self.sources["uniswap_v2"]["router"],
        self.sources["uniswap_v2"]["factory"]
    )
    await uniswap_v2_adapter.load_pools()
```

```
    uniswap_v3_adapter = UniswapV3Adapter(
        self.sources["uniswap_v3"]["router"],
        self.sources["uniswap_v3"]["factory"]
    )
    await uniswap_v3_adapter.load_pools()
```

```
    curve_adapter = CurveAdapter(
        self.sources["curve"]["registry"]
    )
    await curve_adapter.load_pools()
```

```
    for source_id, source in self.sources.items():
        adapter = source["adapter"]
        pools = await adapter.get_pools()
```

```
    for pool in pools:
        self.register_pool(source_id, pool)
```

```
def register_pool(self, source_id, pool):
    pool_key = f'{source_id}:{pool["address"]}'
```

```
    self.pools[pool_key] = {
```

```
"source": source_id,  
"address": pool["address"],  
"tokens": pool["tokens"],  
"reserves": pool["reserves"],  
"fee": pool["fee"]  
}
```

```
pair_key = self._get_pair_key(pool["tokens"][0], pool["tokens"][1])
```

```
if pair_key not in self.token_pairs:  
self.token_pairs[pair_key] = []
```

```
self.token_pairs[pair_key].append(pool_key)
```

```
def get_pools_for_pair(self, token_a, token_b):  
pair_key = self._get_pair_key(token_a, token_b)  
pool_keys = self.token_pairs.get(pair_key, [])
```

```
return [self.pools[key] for key in pool_keys]
```

```
def _get_pair_key(self, token_a, token_b):  
tokens = sorted([token_a.lower(), token_b.lower()])  
return f"{tokens[0]}:{tokens[1]}"
```

```
def update_pool_reserves(self, pool_key, reserves):  
if pool_key in self.pools:  
self.pools[pool_key]["reserves"] = reserves
```

Source Adapters:

```
class SourceAdapter:
```

```
    async def load_pools(self):  
        raise NotImplementedError
```

```
    async def get_pools(self):  
        raise NotImplementedError
```

```
    async def get_quote(self, token_in, token_out, amount_in):
```



```

fn_name = "swapExactTokensForTokens",
args = [
params["amount_in"],
params["min_output"],
params["path"],
params["recipient"],
params["deadline"]
]
)

```

```

class UniswapV3Adapter(SourceAdapter):

```

```

    def __init__(self, router_address, factory_address, web3):
        self.router = router_address
        self.factory = factory_address
        self.web3 = web3
        self.pools = []
        self.quoter = None

```

```

    async def load_pools(self):
        pass

```

```

    async def get_quote(self, token_in, token_out, amount_in,
fee = 3000):
        quoter_contract = self.web3.eth.contract(
            address=self.quoter,
            abi=UNISWAP_V3_QUOTER_ABI
        )

```

```

        amount_out = quoter_contract.functions.quoteExactInputSingle(
            token_in,
            token_out,
            fee,
            amount_in,
            0
        ).call()

```

```

        return amount_out

```



```

def encode_swap(self, params):
    router_contract = self.web3.eth.contract(
        address=self.router,
        abi=UNISWAP_V3_ROUTER_ABI
    )

    return router_contract.encodeABI(
        fn_name="exactInputSingle",
        args=[{
            "tokenIn": params["token_in"],
            "tokenOut": params["token_out"],
            "fee": params["fee"],
            "recipient": params["recipient"],
            "deadline": params["deadline"],
            "amountIn": params["amount_in"],
            "amountOutMinimum": params["min_output"],
            "sqrtPriceLimitX96": 0
        }]
    )

```

```

class CurveAdapter(SourceAdapter):

```

```

    def __init__(self, registry_address, web3):
        self.registry = registry_address
        self.web3 = web3
        self.pools = []

```

```

    async def load_pools(self):
        pass

```

```

    async def get_quote(self, token_in, token_out, amount_in,
        pool_address):
        pool_contract = self.web3.eth.contract(
            address=pool_address,
            abi=CURVE_POOL_ABI
        )

```

```

        i = await self._get_token_index(pool_address, token_in)
        j = await self._get_token_index(pool_address, token_out)

```

```
amount_out = pool_contract.functions.get_dy(i, j, amount_in).call()

return amount_out

async def _get_token_index(self, pool, token):
    pass
```

SECTION 3: PRICE DISCOVERY ENGINE

Query all sources in parallel for best prices.

Price Discovery Engine:

```
class PriceDiscoveryEngine:

    def __init__(self, source_registry):
        self.registry = source_registry
        self.quote_cache = {}
        self.cache_ttl = 10

    async def get_all_quotes(self, token_in, token_out, amount_in):
        pools = self.registry.get_pools_for_pair(token_in, token_out)

        quote_tasks = []

        for pool in pools:
            task = self._get_quote_from_pool(pool, token_in, token_out,
            amount_in)
            quote_tasks.append(task)

        results = await asyncio.gather(*quote_tasks,
        return_exceptions=True)

        quotes = []

        for i, result in enumerate(results):
            if isinstance(result, Exception):
                continue
```

```

if result and result["amount_out"] > 0:
    quotes.append(result)

quotes.sort(key=lambda x: x["amount_out"], reverse=True)

return quotes

async def _get_quote_from_pool(self, pool, token_in, token_out,
amount_in):
    cache_key = f"{pool['address']}:{token_in}:{token_out}:"
    {amount_in}"

    if cache_key in self.quote_cache:
        cached = self.quote_cache[cache_key]
        if time.time() - cached["timestamp"] < self.cache_ttl:
            return cached["quote"]

    source = self.registry.sources[pool["source"]]
    adapter = source["adapter"]

    try:
        amount_out = await adapter.get_quote(token_in, token_out,
        amount_in)

        quote = {
            "source": pool["source"],
            "pool": pool["address"],
            "amount_in": amount_in,
            "amount_out": amount_out,
            "fee": pool["fee"],
            "price": amount_out / amount_in if amount_in > 0 else 0
        }

        self.quote_cache[cache_key] = {
            "quote": quote,
            "timestamp": time.time()
        }

    return quote

```

```
except Exception as e:  
    return None
```

```
async def get_multi_hop_quotes(self, token_in, token_out,  
amount_in, intermediate_tokens):  
    multi_hop_quotes = []
```

```
    for intermediate in intermediate_tokens:  
        if intermediate == token_in or intermediate == token_out:  
            continue
```

```
    first_leg_quotes = await self.get_all_quotes(token_in, intermediate,  
amount_in)
```

```
    if not first_leg_quotes:  
        continue
```

```
    best_first_leg = first_leg_quotes[0]
```

```
    second_leg_quotes = await self.get_all_quotes(  
intermediate, token_out, best_first_leg["amount_out"]  
)
```

```
    if not second_leg_quotes:  
        continue
```

```
    best_second_leg = second_leg_quotes[0]
```

```
    multi_hop_quotes.append({  
"path": [token_in, intermediate, token_out],  
"legs": [best_first_leg, best_second_leg],  
"total_output": best_second_leg["amount_out"],  
"sources": [best_first_leg["source"], best_second_leg["source"]]  
})
```

```
    multi_hop_quotes.sort(key=lambda x: x["total_output"],  
reverse=True)
```

```
    return multi_hop_quotes
```

SECTION 4: AGGREGATOR ROUTER

Find optimal routes considering all factors.

Aggregator Router:

```
class AggregatorRouter:
```

```
    def __init__(self, source_registry):
```

```
        self.registry = source_registry
```

```
        self.price_engine = PriceDiscoveryEngine(source_registry)
```

```
    async def find_optimal_route(self, token_in, token_out, amount_in,
quotes):
```

```
        best_single = await self._find_best_single_route(quotes)
```

```
        best_split = await self._find_best_split_route(
```

```
            token_in, token_out, amount_in, quotes
```

```
        )
```

```
        intermediate_tokens = self._get_common_bases()
```

```
        best_multi_hop = await self._find_best_multi_hop_route(
```

```
            token_in, token_out, amount_in, intermediate_tokens
```

```
        )
```

```
        candidates = []
```

```
        if best_single:
```

```
            candidates.append({
```

```
                "type": "single",
```

```
                "expected_output": best_single["amount_out"],
```

```
                "route": [best_single],
```

```
                "sources": [best_single["source"]],
```

```
                "price_impact": self._calculate_price_impact(
```

```
                    amount_in, best_single["amount_out"], token_in, token_out
```

```
                )
```

```
            })
```

```
if best_split:
    candidates.append({
        "type": "split",
        "expected_output": best_split["total_output"],
        "route": best_split["splits"],
        "sources": list(set(s["source"] for s in best_split["splits"])),
        "price_impact": best_split["price_impact"]
    })
```

```
if best_multi_hop:
    candidates.append({
        "type": "multi_hop",
        "expected_output": best_multi_hop["total_output"],
        "route": best_multi_hop["legs"],
        "sources": best_multi_hop["sources"],
        "price_impact": self._calculate_price_impact(
            amount_in, best_multi_hop["total_output"], token_in, token_out
        )
    })
```

```
if not candidates:
    raise ValueError("No route found")
```

```
candidates.sort(key = lambda x: x["expected_output"],
reverse = True)
```

```
return candidates[0]
```

```
async def _find_best_single_route(self, quotes):
    if not quotes:
        return None
```

```
    return quotes[0]
```

```
async def _find_best_split_route(self, token_in, token_out, amount_in,
quotes):
    if len(quotes) < 2:
        return None
```

```
    top_quotes = quotes[:4]
```

```
best_output = 0
best_split = None
```

```
for allocation in self._generate_allocations(len(top_quotes), 10):
    amounts = [int(amount_in * a / 10) for a in allocation]
```

```
remainder = amount_in - sum(amounts)
amounts[0] += remainder
```

```
splits = []
total_output = 0
```

```
for i, quote in enumerate(top_quotes):
    if amounts[i] == 0:
        continue
```

```
split_quote = await self.price_engine._get_quote_from_pool(
    {"source": quote["source"], "address": quote["pool"], "fee":
    quote["fee"]},
    token_in,
    token_out,
    amounts[i]
)
```

```
if split_quote:
    splits.append({
        **split_quote,
        "split_amount": amounts[i]
    })
    total_output += split_quote["amount_out"]
```

```
if total_output > best_output:
    best_output = total_output
    best_split = {
        "splits": splits,
        "total_output": total_output,
        "price_impact": self._calculate_price_impact(
            amount_in, total_output, token_in, token_out
        )
    }
```

```

}

return best_split

async def _find_best_multi_hop_route(self, token_in, token_out,
amount_in, intermediates):
    multi_hop_quotes = await self.price_engine.get_multi_hop_quotes(
token_in, token_out, amount_in, intermediates
    )

    if not multi_hop_quotes:
        return None

    return multi_hop_quotes[0]

def _get_common_bases(self):
    return [
        "0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2",
        "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48",
        "0xdAC17F958D2ee523a2206206994597C13D831ec7",
        "0x6B175474E89094C44Da98b954EeScdeCB5B7fE40"
    ]

def _generate_allocations(self, n, total):
    if n == 1:
        yield [total]
        return

    for i in range(total + 1):
        for rest in self._generate_allocations(n - 1, total - i):
            yield [i] + rest

def _calculate_price_impact(self, amount_in, amount_out, token_in,
token_out):
    return 0.5

```

SECTION 5: EXECUTION LAYER

Build and execute transactions across multiple DEXes.

Aggregator Executor:

class AggregatorExecutor:

```
def __init__(self, web3, aggregator_contract):
```

```
    self.web3 = web3
```

```
    self.aggregator = aggregator_contract
```

```
    async def build_transaction(self, quote, wallet):
```

```
        route_type = quote["route"][0].get("type", "single")
```

```
        if len(quote["route"]) == 1:
```

```
            return await self._build_single_swap_tx(quote, wallet)
```

```
        else:
```

```
            return await self._build_multi_swap_tx(quote, wallet)
```

```
    async def _build_single_swap_tx(self, quote, wallet):
```

```
        route = quote["route"][0]
```

```
        source = route["source"]
```

```
        adapter = self._get_adapter(source)
```

```
        swap_data = adapter.encode_swap({
```

```
            "token_in": quote["token_in"],
```

```
            "token_out": quote["token_out"],
```

```
            "amount_in": quote["amount_in"],
```

```
            "min_output": quote["min_output"],
```

```
            "recipient": wallet.address,
```

```
            "deadline": int(time.time()) + 1200
```

```
        })
```

```
        return {
```

```
            "to": adapter.router,
```

```
            "data": swap_data,
```

```
            "value": 0 if quote["token_in"] !=
```

```
"0xEeeeeEeeeEeEeeEeEeEeeEEEEEEEEEEEEEEEEEEEE" else
```

```
quote["amount_in"],
```

```
            "gas": 300000,
```

```
            "gasPrice": self.web3.eth.gas_price,
```

```
"nonce": self.web3.eth.get_transaction_count(wallet.address)
}
```

```
async def _build_multi_swap_tx(self, quote, wallet):
    calls = []
```

```
    for i, route in enumerate(quote["route"]):
        adapter = self._get_adapter(route["source"])
```

```
    is_last = i == len(quote["route"]) - 1
    recipient = wallet.address if is_last else self.aggregator
```

```
    if "split_amount" in route:
        amount_in = route["split_amount"]
    else:
        amount_in = route["amount_in"]
```

```
    swap_data = adapter.encode_swap({
        "token_in": route.get("token_in", quote["token_in"]),
        "token_out": route.get("token_out", quote["token_out"]),
        "amount_in": amount_in,
        "min_output": 0 if not is_last else quote["min_output"],
        "recipient": recipient,
        "deadline": int(time.time()) + 1200
    })
```

```
    calls.append({
        "target": adapter.router,
        "callData": swap_data,
        "value": 0
    })
```

```
    multicall_data = self._encode_multicall(calls)
```

```
    return {
        "to": self.aggregator,
        "data": multicall_data,
        "value": 0 if quote["token_in"] !=
        "0xEeeeeEeeeEeEeeEeEeEeEeeEeEeeEeEeeEeE" else
        quote["amount_in"],
```

```
"gas": 150000 * len(calls) + 50000,
"gasPrice": self.web3.eth.gas_price,
"nonce": self.web3.eth.get_transaction_count(wallet.address)
}
```

```
def _encode_multicall(self, calls):
    return b""
```

```
def _get_adapter(self, source):
    pass
```

SECTION 6: GAS OPTIMIZATION

Minimize gas costs for complex routes.

Gas Optimizer:

```
class GasOptimizer:
```

```
    def __init__(self, web3):
        self.web3 = web3
```

```
        self.gas_costs = {
            "uniswap_v2": 120000,
            "uniswap_v3": 150000,
            "sushiswap": 120000,
            "curve": 200000,
            "balancer": 180000
        }
```

```
    async def estimate_gas(self, route):
        total_gas = 21000
```

```
        if isinstance(route["route"], list):
            for leg in route["route"]:
                source = leg.get("source", "uniswap_v2")
                total_gas += self.gas_costs.get(source, 150000)
            else:
                source = route["route"].get("source", "uniswap_v2")
```

```

total_gas += self.gas_costs.get(source, 150000)

if len(route.get("sources", [])) > 1:
    total_gas += 30000

return int(total_gas * 1.1)

async def get_optimal_gas_price(self):
    base_fee = self.web3.eth.get_block("latest").get("baseFeePerGas", 0)

    priority_fee = self.web3.to_wei(2, "gwei")

    return {
        "maxFeePerGas": base_fee * 2 + priority_fee,
        "maxPriorityFeePerGas": priority_fee
    }

def calculate_gas_cost_in_tokens(self, gas_estimate, gas_price,
token_price_eth):
    gas_cost_eth = gas_estimate * gas_price / 10**18
    gas_cost_tokens = gas_cost_eth / token_price_eth
    return gas_cost_tokens

def is_route_profitable_after_gas(self, route, amount_in, gas_price,
token_in_price_eth, token_out_price_eth):
    gas_estimate = route.get("gas_estimate", 200000)
    gas_cost_eth = gas_estimate * gas_price / 10**18

    input_value_eth = amount_in * token_in_price_eth
    output_value_eth = route["expected_output"] * token_out_price_eth

    net_value = output_value_eth - input_value_eth - gas_cost_eth

    return net_value > 0

```

SECTION 7: API DESIGN

RESTful API for the aggregator.

Aggregator API:

```
from aiohttp import web
import json
```

```
class AggregatorAPI:
```

```
    def __init__(self, aggregator):
        self.aggregator = aggregator
        self.app = web.Application()
        self._setup_routes()
```

```
    def _setup_routes(self):
        self.app.router.add_get("/quote", self.handle_quote)
        self.app.router.add_post("/swap", self.handle_swap)
        self.app.router.add_get("/sources", self.handle_sources)
        self.app.router.add_get("/tokens", self.handle_tokens)
```

```
    async def handle_quote(self, request):
        try:
            token_in = request.query.get("tokenIn")
            token_out = request.query.get("tokenOut")
            amount = request.query.get("amount")
            slippage = float(request.query.get("slippage", 0.5))
```

```
            if not all([token_in, token_out, amount]):
                return web.json_response(
                    {"error": "Missing required parameters"},
                    status=400
                )
```

```
            quote = await self.aggregator.get_quote({
                "token_in": token_in,
                "token_out": token_out,
                "amount_in": int(amount),
                "slippage": slippage
            })
```

```
            return web.json_response({
                "tokenIn": quote["token_in"],
```

```
"tokenOut": quote["token_out"],
"amountIn": str(quote["amount_in"]),
"expectedOutput": str(quote["expected_output"]),
"minOutput": str(quote["min_output"]),
"priceImpact": quote["price_impact"],
"gasEstimate": quote["gas_estimate"],
"route": self._format_route(quote["route"]),
"sources": quote["sources"]
})
```

```
except Exception as e:
    return web.json_response(
        {"error": str(e)},
        status=500
    )
```

```
async def handle_swap(self, request):
    try:
        data = await request.json()
```

```
        token_in = data.get("tokenIn")
        token_out = data.get("tokenOut")
        amount = data.get("amount")
        slippage = float(data.get("slippage", 0.5))
        sender = data.get("sender")
```

```
        quote = await self.aggregator.get_quote({
            "token_in": token_in,
            "token_out": token_out,
            "amount_in": int(amount),
            "slippage": slippage
        })
```

```
        tx_data = await self.aggregator.executor.build_transaction(
            quote,
            type("Wallet", (), {"address": sender})()
        )
```

```
    return web.json_response({
        "to": tx_data["to"],
```

```
"data": tx_data["data"].hex() if isinstance(tx_data["data"], bytes) else
tx_data["data"],
"value": str(tx_data["value"]),
"gasLimit": str(tx_data["gas"]),
"quote": {
"expectedOutput": str(quote["expected_output"]),
"minOutput": str(quote["min_output"])
}
})
```

```
except Exception as e:
return web.json_response(
{"error": str(e)},
status = 500
)
```

```
async def handle_sources(self, request):
sources = []
```

```
for source_id, source in self.aggregator.sources.sources.items():
sources.append({
"id": source_id,
"name": source["name"],
"type": source["type"]
})
```

```
return web.json_response({"sources": sources})
```

```
async def handle_tokens(self, request):
tokens = []
```

```
return web.json_response({"tokens": tokens})
```

```
def _format_route(self, route):
if isinstance(route, list):
return [self._format_leg(leg) for leg in route]
return [self._format_leg(route)]
```

```
def _format_leg(self, leg):
return {
```

```
"source": leg.get("source"),
"pool": leg.get("pool"),
"amountIn": str(leg.get("amount_in", 0)),
"amountOut": str(leg.get("amount_out", 0))
}
```

```
def run(self, host="0.0.0.0", port=8080):
web.run_app(self.app, host=host, port=port)
```

SECTION 8: COMPLETE AGGREGATOR

Putting everything together.

Complete DEX Aggregator:

```
class CompleteDEXAggregator:
```

```
def __init__(self, config):
self.config = config
self.web3 = Web3(Web3.HTTPProvider(config.rpc_url))

self.sources = LiquiditySourceRegistry()
self._register_sources()
```

```
self.price_engine = PriceDiscoveryEngine(self.sources)
self.router = AggregatorRouter(self.sources)
self.executor = AggregatorExecutor(self.web3,
config.aggregator_contract)
self.gas_optimizer = GasOptimizer(self.web3)
```

```
self.api = AggregatorAPI(self)
```

```
def _register_sources(self):
self.sources.register_source("uniswap_v2", {
"name": "Uniswap V2",
"type": "amm_v2",
"router_address": self.config.uniswap_v2_router,
"factory_address": self.config.uniswap_v2_factory,
"fee_tiers": [0.3],
```



```
"adapter_class": UniswapV2Adapter
})
```

```
self.sources.register_source("uniswap_v3", {
    "name": "Uniswap V3",
    "type": "amm_v3",
    "router_address": self.config.uniswap_v3_router,
    "factory_address": self.config.uniswap_v3_factory,
    "fee_tiers": [0.01, 0.05, 0.3, 1.0],
    "adapter_class": UniswapV3Adapter
})
```

```
self.sources.register_source("sushiswap", {
    "name": "SushiSwap",
    "type": "amm_v2",
    "router_address": self.config.sushiswap_router,
    "factory_address": self.config.sushiswap_factory,
    "fee_tiers": [0.3],
    "adapter_class": UniswapV2Adapter
})
```

```
self.sources.register_source("curve", {
    "name": "Curve",
    "type": "stable_amm",
    "registry": self.config.curve_registry,
    "adapter_class": CurveAdapter
})
```

```
async def initialize(self):
    await self.sources.load_all_sources()
```

```
async def get_quote(self, params):
    all_quotes = await self.price_engine.get_all_quotes(
        params["token_in"],
        params["token_out"],
        params["amount_in"]
    )
```

```
optimal_route = await self.router.find_optimal_route(
    params["token_in"],
```

```
params["token_out"],
params["amount_in"],
all_quotes
)
```

```
gas_estimate = await
self.gas_optimizer.estimate_gas(optimal_route)
```

```
slippage = params.get("slippage", 0.5)
min_output = int(optimal_route["expected_output"] * (100 -
slippage) / 100)
```

```
return {
"token_in": params["token_in"],
"token_out": params["token_out"],
"amount_in": params["amount_in"],
"expected_output": optimal_route["expected_output"],
"min_output": min_output,
"route": optimal_route["route"],
"sources": optimal_route["sources"],
"gas_estimate": gas_estimate,
"price_impact": optimal_route["price_impact"]
}
```

```
async def execute_swap(self, quote, wallet):
tx = await self.executor.build_transaction(quote, wallet)
```

```
gas_prices = await self.gas_optimizer.get_optimal_gas_price()
tx.update(gas_prices)
```

```
signed = wallet.sign_transaction(tx)
tx_hash =
self.web3.eth.send_raw_transaction(signed.rawTransaction)
```

```
return {
"tx_hash": tx_hash.hex(),
"expected_output": quote["expected_output"],
"min_output": quote["min_output"]
}
```

```
def start_api(self, host="0.0.0.0", port=8080):
    self.api.run(host=host, port=port)
```

```
async def main():
    from dataclasses import dataclass
```

```
@dataclass
class Config:
    rpc_url: str
    uniswap_v2_router: str
    uniswap_v2_factory: str
    uniswap_v3_router: str
    uniswap_v3_factory: str
    sushiswap_router: str
    sushiswap_factory: str
    curve_registry: str
    aggregator_contract: str
```

```
config = Config(
    rpc_url="https://mainnet.infura.io/v3/YOUR_KEY",
    uniswap_v2_router="0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D",
    uniswap_v2_factory="0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f",
    uniswap_v3_router="0xE592427A0AEce92De3Edee1F18E0157C05861564",
    uniswap_v3_factory="0x1F98431c8aD98523631AE4a59f267346ea31F984",
    sushiswap_router="0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F",
    sushiswap_factory="0xC0AEe478e3658e2610c5F7A4A2E1777cE9e4f2Ac",
    curve_registry="0x90E00ACe148ca3b23Ac1bC8C240C2a7Dd9c2d7f5",
    aggregator_contract="0x..."
)
```

```
aggregator = CompleteDEXAggregator(config)
await aggregator.initialize()
```

```
aggregator.start_api()
```

CHAPTER SUMMARY

This chapter covered DEX aggregation:

Aggregator architecture combines liquidity sources, price discovery, routing, and execution into a unified system.

Liquidity source registry tracks all DEXes, pools, and their adapters for unified access.

Price discovery queries all sources in parallel to find the best quotes.

Aggregator routing finds optimal execution across single routes, split routes, and multi-hop paths.

Execution layer builds and executes transactions across multiple DEXes efficiently.

Gas optimization minimizes costs and ensures trades remain profitable after fees.

API design provides RESTful endpoints for quotes, swaps, and source information.

Next chapter: Building Your Own DEX. We implement a complete decentralized exchange.

CHAPTER 8: BUILDING YOUR OWN DEX

Everything learned in this book comes together here. We build a complete decentralized exchange - factory contract, pair contract, router, frontend, and deployment. A functional DEX from scratch.

SECTION 1: DEX ARCHITECTURE OVERVIEW

A complete DEX requires multiple components working together.

System Components:

Factory Contract: Creates and tracks trading pairs

Pair Contract: Holds liquidity and executes swaps

Router Contract: User-facing swap and liquidity functions

Frontend: Web interface for users

Indexer: Tracks events and provides analytics

API: Serves data to frontend

DEX Core Design:

```
class DEXCore:
```

```
    def __init__(self):
        self.factory = None
        self.router = None
        self.pairs = {}
        self.token_registry = {}
```

```
    def deploy(self, deployer):
        self.factory = self.deploy_factory(deployer)
        self.router = self.deploy_router(deployer, self.factory.address)
```

```
    return {
        "factory": self.factory.address,
        "router": self.router.address
    }
```

```
    def deploy_factory(self, deployer):
        pass
```

```
    def deploy_router(self, deployer, factory_address):
        pass
```

SECTION 2: PAIR CONTRACT

The pair contract holds reserves and executes swaps. This is the heart of the DEX.

Pair Contract Implementation:

```
class PairContract:
```

```
    MINIMUM_LIQUIDITY = 1000
```

```
def __init__(self, token0, token1, factory):
    self.token0 = token0
    self.token1 = token1
    self.factory = factory
```

```
self.reserve0 = 0
self.reserve1 = 0
self.block_timestamp_last = 0
```

```
self.price0_cumulative_last = 0
self.price1_cumulative_last = 0
```

```
self.total_supply = 0
self.balances = {}
```

```
self.k_last = 0
self.unlocked = True
```

```
def get_reserves(self):
    return self.reserve0, self.reserve1, self.block_timestamp_last
```

```
def _update(self, balance0, balance1, reserve0, reserve1,
timestamp):
    time_elapsed = timestamp - self.block_timestamp_last
```

```
    if time_elapsed > 0 and reserve0 > 0 and reserve1 > 0:
        self.price0_cumulative_last += (reserve1 << 112) // reserve0 *
time_elapsed
        self.price1_cumulative_last += (reserve0 << 112) // reserve1 *
time_elapsed
```

```
self.reserve0 = balance0
self.reserve1 = balance1
self.block_timestamp_last = timestamp
```

```
def _mint_fee(self, reserve0, reserve1):
    fee_to = self.factory.fee_to
```

```
    if fee_to:
```

```

if self.k_last != 0:
    from math import sqrt

    root_k = int(sqrt(reserve0 * reserve1))
    root_k_last = int(sqrt(self.k_last))

    if root_k > root_k_last:
        numerator = self.total_supply * (root_k - root_k_last)
        denominator = root_k * 5 + root_k_last
        liquidity = numerator // denominator

    if liquidity > 0:
        self._mint(fee_to, liquidity)

    else:
        if self.k_last != 0:
            self.k_last = 0

def mint(self, to, timestamp):
    if not self.unlocked:
        raise ValueError("Locked")
    self.unlocked = False

    reserve0, reserve1, _ = self.get_reserves()

    balance0 = self._get_balance(self.token0)
    balance1 = self._get_balance(self.token1)

    amount0 = balance0 - reserve0
    amount1 = balance1 - reserve1

    self._mint_fee(reserve0, reserve1)

    if self.total_supply == 0:
        from math import sqrt
        liquidity = int(sqrt(amount0 * amount1)) -
        self.MINIMUM_LIQUIDITY

    self._mint("0x0", self.MINIMUM_LIQUIDITY)
    else:

```

```

liquidity = min(
amount0 * self.total_supply // reserve0,
amount1 * self.total_supply // reserve1
)

if liquidity <= 0:
raise ValueError("Insufficient liquidity minted")

self._mint(to, liquidity)

self._update(balance0, balance1, reserve0, reserve1, timestamp)

if self.factory.fee_to:
self.k_last = self.reserve0 * self.reserve1

self.unlocked = True

return liquidity

def burn(self, to, timestamp):
if not self.unlocked:
raise ValueError("Locked")
self.unlocked = False

reserve0, reserve1, _ = self.get_reserves()

balance0 = self._get_balance(self.token0)
balance1 = self._get_balance(self.token1)

liquidity = self.balances.get(self.address, 0)

self._mint_fee(reserve0, reserve1)

amount0 = liquidity * balance0 // self.total_supply
amount1 = liquidity * balance1 // self.total_supply

if amount0 <= 0 or amount1 <= 0:
raise ValueError("Insufficient liquidity burned")

self._burn(self.address, liquidity)

```



```

self._transfer_token(self.token0, to, amount0)
self._transfer_token(self.token1, to, amount1)

balance0 = self._get_balance(self.token0)
balance1 = self._get_balance(self.token1)

self._update(balance0, balance1, reserve0, reserve1, timestamp)

if self.factory.fee_to:
    self.k_last = self.reserve0 * self.reserve1

self.unlocked = True

return amount0, amount1

def swap(self, amount0_out, amount1_out, to, data, timestamp):
    if not self.unlocked:
        raise ValueError("Locked")
    self.unlocked = False

    if amount0_out <= 0 and amount1_out <= 0:
        raise ValueError("Insufficient output amount")

    reserve0, reserve1, _ = self.get_reserves()

    if amount0_out >= reserve0 or amount1_out >= reserve1:
        raise ValueError("Insufficient liquidity")

    if to == self.token0 or to == self.token1:
        raise ValueError("Invalid to")

    if amount0_out > 0:
        self._transfer_token(self.token0, to, amount0_out)
    if amount1_out > 0:
        self._transfer_token(self.token1, to, amount1_out)

    if data:
        self._flash_callback(to, amount0_out, amount1_out, data)

```

```
balance0 = self._get_balance(self.token0)
```

```
balance1 = self._get_balance(self.token1)
```

```
amount0_in = balance0 - (reserve0 - amount0_out) if balance0 >
reserve0 - amount0_out else 0
```

```
amount1_in = balance1 - (reserve1 - amount1_out) if balance1 >
reserve1 - amount1_out else 0
```

```
if amount0_in <= 0 and amount1_in <= 0:
```

```
raise ValueError("Insufficient input amount")
```

```
balance0_adjusted = balance0 * 1000 - amount0_in * 3
```

```
balance1_adjusted = balance1 * 1000 - amount1_in * 3
```

```
if balance0_adjusted * balance1_adjusted < reserve0 * reserve1 *
1000000:
```

```
raise ValueError("K")
```

```
self._update(balance0, balance1, reserve0, reserve1, timestamp)
```

```
self.unlocked = True
```

```
def _mint(self, to, amount):
```

```
self.total_supply += amount
```

```
if to not in self.balances:
```

```
self.balances[to] = 0
```

```
self.balances[to] += amount
```

```
def _burn(self, from_addr, amount):
```

```
self.balances[from_addr] -= amount
```

```
self.total_supply -= amount
```

```
def _get_balance(self, token):
```

```
return 0
```

```
def _transfer_token(self, token, to, amount):
```

```
pass
```

```
def _flash_callback(self, to, amount0, amount1, data):
```

```
pass
```

SECTION 3: FACTORY CONTRACT

The factory creates pairs and maintains the registry.

Factory Contract Implementation:

```
class FactoryContract:
```

```
    def __init__(self, fee_to_setter):
        self.fee_to_setter = fee_to_setter
        self.fee_to = None
```

```
        self.pairs = {}
        self.all_pairs = []
```

```
    def create_pair(self, token_a, token_b):
        if token_a == token_b:
            raise ValueError("Identical addresses")
```

```
        token0, token1 = (token_a, token_b) if token_a < token_b else
        (token_b, token_a)
```

```
        if not token0:
            raise ValueError("Zero address")
```

```
        pair_key = f"{token0}:{token1}"
```

```
        if pair_key in self.pairs:
            raise ValueError("Pair exists")
```

```
        pair = PairContract(token0, token1, self)
        pair.address = self._generate_pair_address(token0, token1)
```

```
        self.pairs[pair_key] = pair.address
        self.all_pairs.append(pair.address)
```

```
    return pair.address
```

```

def get_pair(self, token_a, token_b):
    token0, token1 = (token_a, token_b) if token_a < token_b else
(token_b, token_a)
    pair_key = f"{token0}:{token1}"

    return self.pairs.get(pair_key)

def all_pairs_length(self):
    return len(self.all_pairs)

def set_fee_to(self, fee_to, sender):
    if sender != self.fee_to_setter:
        raise ValueError("Forbidden")

    self.fee_to = fee_to

def set_fee_to_setter(self, new_setter, sender):
    if sender != self.fee_to_setter:
        raise ValueError("Forbidden")

    self.fee_to_setter = new_setter

def _generate_pair_address(self, token0, token1):
    import hashlib

    data = f"{token0}{token1}{len(self.all_pairs)}"
    return "0x" + hashlib.sha256(data.encode()).hexdigest()[:40]

```

SECTION 4: ROUTER CONTRACT

The router provides user-friendly functions for swapping and liquidity.

Router Contract Implementation:

```

class RouterContract:

    def __init__(self, factory, weth):
        self.factory = factory

```

```
self.weth = weth
```

```
def add_liquidity(  
    self,  
    token_a,  
    token_b,  
    amount_a_desired,  
    amount_b_desired,  
    amount_a_min,  
    amount_b_min,  
    to,  
    deadline,  
    timestamp  
):  
    if timestamp > deadline:  
        raise ValueError("Expired")
```

```
    amount_a, amount_b = self._add_liquidity(  
        token_a, token_b,  
        amount_a_desired, amount_b_desired,  
        amount_a_min, amount_b_min  
    )
```

```
    pair = self._pair_for(token_a, token_b)
```

```
    self._transfer_from(token_a, msg_sender, pair, amount_a)  
    self._transfer_from(token_b, msg_sender, pair, amount_b)
```

```
    liquidity = pair.mint(to, timestamp)
```

```
    return amount_a, amount_b, liquidity
```

```
def add_liquidity_eth(  
    self,  
    token,  
    amount_token_desired,  
    amount_token_min,  
    amount_eth_min,  
    to,  
    deadline,
```

```
timestamp,  
value  
):  
if timestamp > deadline:  
    raise ValueError("Expired")
```

```
amount_token, amount_eth = self._add_liquidity(  
    token, self.weth,  
    amount_token_desired, value,  
    amount_token_min, amount_eth_min  
)
```

```
pair = self._pair_for(token, self.weth)
```

```
self._transfer_from(token, msg_sender, pair, amount_token)  
self._wrap_eth(amount_eth)  
self._transfer(self.weth, pair, amount_eth)
```

```
liquidity = pair.mint(to, timestamp)
```

```
if value > amount_eth:  
    self._transfer_eth(msg_sender, value - amount_eth)
```

```
return amount_token, amount_eth, liquidity
```

```
def remove_liquidity(  
    self,  
    token_a,  
    token_b,  
    liquidity,  
    amount_a_min,  
    amount_b_min,  
    to,  
    deadline,  
    timestamp  
):  
    if timestamp > deadline:  
        raise ValueError("Expired")
```

```
pair = self._pair_for(token_a, token_b)
```

```
pair.balances[pair.address] = liquidity
```

```
amount0, amount1 = pair.burn(to, timestamp)
```

```
token0, _ = self._sort_tokens(token_a, token_b)
```

```
if token_a == token0:
```

```
    amount_a, amount_b = amount0, amount1
```

```
else:
```

```
    amount_a, amount_b = amount1, amount0
```

```
if amount_a < amount_a_min:
```

```
    raise ValueError("Insufficient A amount")
```

```
if amount_b < amount_b_min:
```

```
    raise ValueError("Insufficient B amount")
```

```
return amount_a, amount_b
```

```
def swap_exact_tokens_for_tokens(
```

```
    self,
```

```
    amount_in,
```

```
    amount_out_min,
```

```
    path,
```

```
    to,
```

```
    deadline,
```

```
    timestamp
```

```
):
```

```
    if timestamp > deadline:
```

```
        raise ValueError("Expired")
```

```
amounts = self.get_amounts_out(amount_in, path)
```

```
if amounts[-1] < amount_out_min:
```

```
    raise ValueError("Insufficient output amount")
```

```
self._transfer_from(path[0], msg_sender, self._pair_for(path[0],  
path[1]), amounts[0])
```

```
self._swap(amounts, path, to, timestamp)
```

```
return amounts
```

```
def swap_tokens_for_exact_tokens(  
    self,  
    amount_out,  
    amount_in_max,  
    path,  
    to,  
    deadline,  
    timestamp  
):  
    if timestamp > deadline:  
        raise ValueError("Expired")
```

```
    amounts = self.get_amounts_in(amount_out, path)
```

```
    if amounts[0] > amount_in_max:  
        raise ValueError("Excessive input amount")
```

```
    self._transfer_from(path[0], msg_sender, self._pair_for(path[0],  
path[1]), amounts[0])
```

```
    self.swap(amounts, path, to, timestamp)
```

```
return amounts
```

```
def swap_exact_eth_for_tokens(  
    self,  
    amount_out_min,  
    path,  
    to,  
    deadline,  
    timestamp,  
    value  
):  
    if timestamp > deadline:  
        raise ValueError("Expired")
```

```
    if path[0] != self.weth:
```



```
raise ValueError("Invalid path")
```

```
amounts = self.get_amounts_out(value, path)
```

```
if amounts[-1] < amount_out_min:  
raise ValueError("Insufficient output amount")
```

```
self._wrap_eth(amounts[0])  
self._transfer(self.weth, self._pair_for(path[0], path[1]),  
amounts[0])
```

```
self._swap(amounts, path, to, timestamp)
```

```
return amounts
```

```
def swap_exact_tokens_for_eth(  
self,  
amount_in,  
amount_out_min,  
path,  
to,  
deadline,  
timestamp  
):  
if timestamp > deadline:  
raise ValueError("Expired")
```

```
if path[-1] != self.weth:  
raise ValueError("Invalid path")
```

```
amounts = self.get_amounts_out(amount_in, path)
```

```
if amounts[-1] < amount_out_min:  
raise ValueError("Insufficient output amount")
```

```
self._transfer_from(path[0], msg_sender, self._pair_for(path[0],  
path[1]), amounts[0])
```

```
self._swap(amounts, path, self.address, timestamp)
```

```
self._unwrap_eth(amounts[-1])
self._transfer_eth(to, amounts[-1])
```

```
return amounts
```

```
def _add_liquidity(
    self,
    token_a,
    token_b,
    amount_a_desired,
    amount_b_desired,
    amount_a_min,
    amount_b_min
):
    pair = self.factory.get_pair(token_a, token_b)
```

```
    if not pair:
        pair = self.factory.create_pair(token_a, token_b)
```

```
    reserve_a, reserve_b, _ = pair.get_reserves()
```

```
    if reserve_a == 0 and reserve_b == 0:
        return amount_a_desired, amount_b_desired
```

```
    amount_b_optimal = self._quote(amount_a_desired, reserve_a,
    reserve_b)
```

```
    if amount_b_optimal <= amount_b_desired:
        if amount_b_optimal < amount_b_min:
            raise ValueError("Insufficient B amount")
        return amount_a_desired, amount_b_optimal
```

```
    amount_a_optimal = self._quote(amount_b_desired, reserve_b,
    reserve_a)
```

```
    if amount_a_optimal > amount_a_desired:
        raise ValueError("Calculation error")
```

```
    if amount_a_optimal < amount_a_min:
        raise ValueError("Insufficient A amount")
```

```
return amount_a_optimal, amount_b_desired
```

```
def _swap(self, amounts, path, to, timestamp):  
    for i in range(len(path) - 1):  
        token_in, token_out = path[i], path[i + 1]  
        token0, _ = self._sort_tokens(token_in, token_out)
```

```
        amount_out = amounts[i + 1]
```

```
        if token_in == token0:  
            amount0_out, amount1_out = 0, amount_out  
        else:  
            amount0_out, amount1_out = amount_out, 0
```

```
        next_to = to if i == len(path) - 2 else self._pair_for(path[i + 1],  
path[i + 2])
```

```
        pair = self._pair_for(token_in, token_out)  
        pair.swap(amount0_out, amount1_out, next_to, None, timestamp)
```

```
def get_amounts_out(self, amount_in, path):  
    if len(path) < 2:  
        raise ValueError("Invalid path")
```

```
    amounts = [amount_in]
```

```
    for i in range(len(path) - 1):  
        pair = self._pair_for(path[i], path[i + 1])  
        reserve_in, reserve_out, _ = pair.get_reserves()
```

```
        amount_out = self._get_amount_out(amounts[i], reserve_in,  
reserve_out)  
        amounts.append(amount_out)
```

```
    return amounts
```

```
def get_amounts_in(self, amount_out, path):  
    if len(path) < 2:  
        raise ValueError("Invalid path")
```

```
amounts = [0] * len(path)
amounts[-1] = amount_out
```

```
for i in range(len(path) - 1, 0, -1):
    pair = self._pair_for(path[i - 1], path[i])
    reserve_in, reserve_out, _ = pair.get_reserves()
```

```
    amount_in = self._get_amount_in(amounts[i], reserve_in,
    reserve_out)
    amounts[i - 1] = amount_in
```

```
return amounts
```

```
def _get_amount_out(self, amount_in, reserve_in, reserve_out):
    if amount_in <= 0:
        raise ValueError("Insufficient input amount")
    if reserve_in <= 0 or reserve_out <= 0:
        raise ValueError("Insufficient liquidity")
```

```
    amount_in_with_fee = amount_in * 997
    numerator = amount_in_with_fee * reserve_out
    denominator = reserve_in * 1000 + amount_in_with_fee
```

```
    return numerator // denominator
```

```
def _get_amount_in(self, amount_out, reserve_in, reserve_out):
    if amount_out <= 0:
        raise ValueError("Insufficient output amount")
    if reserve_in <= 0 or reserve_out <= 0:
        raise ValueError("Insufficient liquidity")
```

```
    numerator = reserve_in * amount_out * 1000
    denominator = (reserve_out - amount_out) * 997
```

```
    return numerator // denominator + 1
```

```
def _quote(self, amount_a, reserve_a, reserve_b):
    if amount_a <= 0:
        raise ValueError("Insufficient amount")
```

```

if reserve_a <= 0 or reserve_b <= 0:
    raise ValueError("Insufficient liquidity")

return amount_a * reserve_b // reserve_a

def _sort_tokens(self, token_a, token_b):
    if token_a < token_b:
        return token_a, token_b
    return token_b, token_a

def _pair_for(self, token_a, token_b):
    return self.factory.get_pair(token_a, token_b)

def _transfer_from(self, token, from_addr, to, amount):
    pass

def _transfer(self, token, to, amount):
    pass

def _wrap_eth(self, amount):
    pass

def _unwrap_eth(self, amount):
    pass

def _transfer_eth(self, to, amount):
    pass

```

SECTION 5: EVENT INDEXER

Track all DEX events for analytics and UI.

Event Indexer:

```

class DEXIndexer:

    def __init__(self, web3, factory_address, start_block):
        self.web3 = web3
        self.factory = factory_address

```

```
self.start_block = start_block
```

```
self.pairs = {}
```

```
self.swaps = []
```

```
self.liquidity_events = []
```

```
self.current_block = start_block
```

```
async def start(self):
```

```
    await self.index_historical()
```

```
    asyncio.create_task(self.listen_new_events())
```

```
async def index_historical(self):
```

```
    current = self.web3.eth.block_number
```

```
    for block in range(self.start_block, current, 1000):
```

```
        end_block = min(block + 999, current)
```

```
        await self.process_block_range(block, end_block)
```

```
    self.current_block = current
```

```
async def process_block_range(self, from_block, to_block):
```

```
    pair_created_events = await self._get_events(
```

```
        self.factory,
```

```
        "PairCreated",
```

```
        from_block,
```

```
        to_block
```

```
    )
```

```
    for event in pair_created_events:
```

```
        await self.handle_pair_created(event)
```

```
    for pair_address in self.pairs.keys():
```

```
        swap_events = await self._get_events(
```

```
            pair_address,
```

```
            "Swap",
```

```
            from_block,
```

```
            to_block
```

```
        )
```

```
for event in swap_events:
    await self.handle_swap(event)
```

```
mint_events = await self._get_events(
    pair_address,
    "Mint",
    from_block,
    to_block
)
```

```
for event in mint_events:
    await self.handle_mint(event)
```

```
burn_events = await self._get_events(
    pair_address,
    "Burn",
    from_block,
    to_block
)
```

```
for event in burn_events:
    await self.handle_burn(event)
```

```
async def listen_new_events(self):
    while True:
        current = self.web3.eth.block_number
```

```
        if current > self.current_block:
            await self.process_block_range(self.current_block + 1, current)
            self.current_block = current
```

```
        await asyncio.sleep(12)
```

```
async def handle_pair_created(self, event):
    pair_address = event["args"]["pair"]
    token0 = event["args"]["token0"]
    token1 = event["args"]["token1"]
```

```
    self.pairs[pair_address] = {
        "address": pair_address,
```

```
"token0": token0,  
"token1": token1,  
"created_block": event["blockNumber"],  
"total_volume_0": 0,  
"total_volume_1": 0,  
"reserve0": 0,  
"reserve1": 0  
}
```

```
async def handle_swap(self, event):  
    pair_address = event["address"]
```

```
    swap = {  
        "pair": pair_address,  
        "sender": event["args"]["sender"],  
        "amount0_in": event["args"]["amount0In"],  
        "amount1_in": event["args"]["amount1In"],  
        "amount0_out": event["args"]["amount0Out"],  
        "amount1_out": event["args"]["amount1Out"],  
        "to": event["args"]["to"],  
        "block": event["blockNumber"],  
        "tx_hash": event["transactionHash"].hex()  
    }
```

```
    self.swaps.append(swap)
```

```
    if pair_address in self.pairs:  
        self.pairs[pair_address]["total_volume_0"] += swap["amount0_in"]  
        + swap["amount0_out"]  
        self.pairs[pair_address]["total_volume_1"] += swap["amount1_in"]  
        + swap["amount1_out"]
```

```
    async def handle_mint(self, event):  
        pair_address = event["address"]
```

```
        mint = {  
            "type": "mint",  
            "pair": pair_address,  
            "sender": event["args"]["sender"],  
            "amount0": event["args"]["amount0"],
```



```
"amount1": event["args"]["amount1"],
"block": event["blockNumber"],
"tx_hash": event["transactionHash"].hex()
}
```

```
self.liquidity_events.append(mint)
```

```
async def handle_burn(self, event):
    pair_address = event["address"]
```

```
    burn = {
        "type": "burn",
        "pair": pair_address,
        "sender": event["args"]["sender"],
        "amount0": event["args"]["amount0"],
        "amount1": event["args"]["amount1"],
        "to": event["args"]["to"],
        "block": event["blockNumber"],
        "tx_hash": event["transactionHash"].hex()
    }
```

```
self.liquidity_events.append(burn)
```

```
async def _get_events(self, address, event_name, from_block,
to_block):
    return []
```

```
def get_pair_stats(self, pair_address):
    return self.pairs.get(pair_address)
```

```
def get_recent_swaps(self, pair_address=None, limit=100):
    if pair_address:
        swaps = [s for s in self.swaps if s["pair"] == pair_address]
    else:
        swaps = self.swaps
```

```
    return swaps[-limit:]
```

```
def get_volume_24h(self, pair_address):
    pass
```

SECTION 6: FRONTEND INTERFACE

Web interface for users to interact with the DEX.

DEX Frontend Service:

```
class DEXFrontendService:
```

```
    def __init__(self, web3, router_address, factory_address, indexer):
        self.web3 = web3
        self.router_address = router_address
        self.factory_address = factory_address
        self.indexer = indexer
```

```
    async def get_swap_quote(self, token_in, token_out, amount_in):
        path = [token_in, token_out]
```

```
        router = self._get_router_contract()
```

```
        try:
            amounts = router.functions.getAmountsOut(amount_in,
path).call()
```

```
            reserve_in, reserve_out = await self._get_reserves(token_in,
token_out)
            spot_price = reserve_out / reserve_in if reserve_in > 0 else 0
```

```
            execution_price = amounts[-1] / amount_in if amount_in > 0 else
0
            price_impact = (spot_price - execution_price) / spot_price * 100 if
spot_price > 0 else 0
```

```
        return {
            "amount_in": amount_in,
            "amount_out": amounts[-1],
            "price_impact": price_impact,
            "path": path,
            "execution_price": execution_price
```

```
}
```

```
except Exception as e:  
    return {"error": str(e)}
```

```
async def build_swap_transaction(self, params, wallet_address):  
    router = self._get_router_contract()
```

```
    deadline = int(time.time()) + 1200  
    min_output = int(params["amount_out"] * (100 -  
params["slippage"])) / 100)
```

```
    if params["token_in"] ==  
"0xEeeeeEeeeEeEeeEeEeEeEeEeEeEeEeEeEeEeEeE":  
        tx = router.functions.swapExactETHForTokens(  
            min_output,  
            params["path"],  
            wallet_address,  
            deadline  
        ).build_transaction({  
            "from": wallet_address,  
            "value": params["amount_in"],  
            "gas": 300000,  
            "gasPrice": self.web3.eth.gas_price,  
            "nonce": self.web3.eth.get_transaction_count(wallet_address)  
        })
```

```
    elif params["token_out"] ==  
"0xEeeeeEeeeEeEeeEeEeEeEeEeEeEeEeEeEeEeEeE":  
        tx = router.functions.swapExactTokensForETH(  
            params["amount_in"],  
            min_output,  
            params["path"],  
            wallet_address,  
            deadline  
        ).build_transaction({  
            "from": wallet_address,  
            "gas": 300000,  
            "gasPrice": self.web3.eth.gas_price,  
            "nonce": self.web3.eth.get_transaction_count(wallet_address)
```

```
})
```

```
else:
```

```
tx = router.functions.swapExactTokensForTokens(  
    params["amount_in"],  
    min_output,  
    params["path"],  
    wallet_address,  
    deadline  
)  
.build_transaction({  
    "from": wallet_address,  
    "gas": 300000,  
    "gasPrice": self.web3.eth.gas_price,  
    "nonce": self.web3.eth.get_transaction_count(wallet_address)  
})
```

```
return tx
```

```
async def get_liquidity_quote(self, token_a, token_b, amount_a):  
    reserve_a, reserve_b = await self._get_reserves(token_a, token_b)
```

```
    if reserve_a == 0 or reserve_b == 0:
```

```
        return {  
            "amount_a": amount_a,  
            "amount_b": None,  
            "share": 100,  
            "new_pool": True  
        }
```

```
    amount_b = amount_a * reserve_b // reserve_a
```

```
    pool_value = reserve_a * 2
```

```
    share = amount_a * 2 / (pool_value + amount_a * 2) * 100
```

```
    return {  
        "amount_a": amount_a,  
        "amount_b": amount_b,  
        "share": share,  
        "new_pool": False,  
        "reserve_a": reserve_a,
```

```
"reserve_b": reserve_b
}
```

```
async def build_add_liquidity_transaction(self, params,
wallet_address):
```

```
    router = self._get_router_contract()
```

```
    deadline = int(time.time()) + 1200
```

```
    slippage = params.get("slippage", 0.5)
```

```
    min_a = int(params["amount_a"] * (100 - slippage) / 100)
```

```
    min_b = int(params["amount_b"] * (100 - slippage) / 100)
```

```
    tx = router.functions.addLiquidity(
```

```
        params["token_a"],
```

```
        params["token_b"],
```

```
        params["amount_a"],
```

```
        params["amount_b"],
```

```
        min_a,
```

```
        min_b,
```

```
        wallet_address,
```

```
        deadline
```

```
    ).build_transaction({
```

```
        "from": wallet_address,
```

```
        "gas": 400000,
```

```
        "gasPrice": self.web3.eth.gas_price,
```

```
        "nonce": self.web3.eth.get_transaction_count(wallet_address)
```

```
    })
```

```
    return tx
```

```
async def get_pool_info(self, token_a, token_b):
```

```
    pair_address = await self._get_pair_address(token_a, token_b)
```

```
    if not pair_address:
```

```
        return None
```

```
    reserve_a, reserve_b = await self._get_reserves(token_a, token_b)
```

```
    stats = self.indexer.get_pair_stats(pair_address)
```

```

return {
    "pair_address": pair_address,
    "token_a": token_a,
    "token_b": token_b,
    "reserve_a": reserve_a,
    "reserve_b": reserve_b,
    "total_volume_a": stats["total_volume_0"] if stats else 0,
    "total_volume_b": stats["total_volume_1"] if stats else 0
}

```

```

async def get_all_pools(self):
    pools = []

```

```

    for pair_address, pair_info in self.indexer.pairs.items():
        reserve_a, reserve_b = await self._get_reserves(
            pair_info["token0"],
            pair_info["token1"]
        )

```

```

        pools.append({
            "address": pair_address,
            "token0": pair_info["token0"],
            "token1": pair_info["token1"],
            "reserve0": reserve_a,
            "reserve1": reserve_b,
            "volume_24h": 0
        })

```

```

    return pools

```

```

async def _get_reserves(self, token_a, token_b):
    pair_address = await self._get_pair_address(token_a, token_b)

```

```

    if not pair_address:
        return 0, 0

```

```

    pair_contract = self.web3.eth.contract(
        address=pair_address,
        abi=PAIR_ABI
    )

```

)

```
reserves = pair_contract.functions.getReserves().call()
```

```
return reserves[0], reserves[1]
```

```
async def _get_pair_address(self, token_a, token_b):  
    factory = self.web3.eth.contract(  
        address=self.factory_address,  
        abi=FACTORY_ABI  
    )
```

```
return factory.functions.getPair(token_a, token_b).call()
```

```
def _get_router_contract(self):  
    return self.web3.eth.contract(  
        address=self.router_address,  
        abi=ROUTER_ABI  
    )
```

SECTION 7: DEPLOYMENT

Deploy the complete DEX to a blockchain.

DEX Deployer:

```
class DEXDeployer:
```

```
    def __init__(self, web3, deployer_wallet):  
        self.web3 = web3  
        self.wallet = deployer_wallet
```

```
    async def deploy_full_dex(self, weth_address):  
        factory_address = await self.deploy_factory()
```

```
        router_address = await self.deploy_router(factory_address,  
        weth_address)
```

```
    return {
```

```
"factory": factory_address,  
"router": router_address,  
"weth": weth_address  
}
```

```
async def deploy_factory(self):  
    factory_bytecode = self._get_factory_bytecode()  
    factory_abi = self._get_factory_abi()
```

```
    contract = self.web3.eth.contract(abi=factory_abi,  
    bytecode=factory_bytecode)
```

```
    tx = contract.constructor(self.wallet.address).build_transaction({  
        "from": self.wallet.address,  
        "gas": 5000000,  
        "gasPrice": self.web3.eth.gas_price,  
        "nonce": self.web3.eth.get_transaction_count(self.wallet.address)  
    })
```

```
    signed = self.wallet.sign_transaction(tx)  
    tx_hash =  
    self.web3.eth.send_raw_transaction(signed.rawTransaction)
```

```
    receipt = self.web3.eth.wait_for_transaction_receipt(tx_hash)
```

```
    return receipt["contractAddress"]
```

```
async def deploy_router(self, factory_address, weth_address):  
    router_bytecode = self._get_router_bytecode()  
    router_abi = self._get_router_abi()
```

```
    contract = self.web3.eth.contract(abi=router_abi,  
    bytecode=router_bytecode)
```

```
    tx = contract.constructor(factory_address,  
    weth_address).build_transaction({  
        "from": self.wallet.address,  
        "gas": 5000000,  
        "gasPrice": self.web3.eth.gas_price,  
        "nonce": self.web3.eth.get_transaction_count(self.wallet.address)
```



```
})
```

```
signed = self.wallet.sign_transaction(tx)
tx_hash =
self.web3.eth.send_raw_transaction(signed.rawTransaction)
```

```
receipt = self.web3.eth.wait_for_transaction_receipt(tx_hash)
```

```
return receipt["contractAddress"]
```

```
async def create_initial_pairs(self, factory_address, token_pairs):
factory = self.web3.eth.contract(
address=factory_address,
abi=self._get_factory_abi()
)
```

```
created_pairs = []
```

```
for token_a, token_b in token_pairs:
tx = factory.functions.createPair(token_a,
token_b).build_transaction({
"from": self.wallet.address,
"gas": 3000000,
"gasPrice": self.web3.eth.gas_price,
"nonce": self.web3.eth.get_transaction_count(self.wallet.address)
})
```

```
signed = self.wallet.sign_transaction(tx)
tx_hash =
self.web3.eth.send_raw_transaction(signed.rawTransaction)
```

```
receipt = self.web3.eth.wait_for_transaction_receipt(tx_hash)
```

```
pair_address = factory.functions.getPair(token_a, token_b).call()
```

```
created_pairs.append({
"token_a": token_a,
"token_b": token_b,
"pair": pair_address
})
```

```
return created_pairs
```

```
def _get_factory_bytecode(self):  
    return ""
```

```
def _get_factory_abi(self):  
    return []
```

```
def _get_router_bytecode(self):  
    return ""
```

```
def _get_router_abi(self):  
    return []
```

SECTION 8: COMPLETE DEX SYSTEM

Putting everything together.

Complete DEX System:

```
class CompleteDEX:
```

```
    def __init__(self, config):  
        self.config = config  
        self.web3 = Web3(Web3.HTTPProvider(config.rpc_url))
```

```
        self.factory_address = config.factory_address  
        self.router_address = config.router_address
```

```
        self.indexer = DEXIndexer(  
            self.web3,  
            self.factory_address,  
            config.start_block  
        )
```

```
        self.frontend_service = DEXFrontendService(  
            self.web3,  
            self.router_address,
```

```
self.factory_address,  
self.indexer  
)
```

```
self.api = DEXAPI(self)
```

```
async def start(self):  
    await self.indexer.start()
```

```
print(f'DEX System Started')  
print(f'Factory: {self.factory_address}')  
print(f'Router: {self.router_address}')
```

```
self.api.run()
```

```
async def get_quote(self, token_in, token_out, amount_in):  
    return await self.frontend_service.get_swap_quote(  
        token_in, token_out, amount_in  
    )
```

```
async def build_swap_tx(self, params, wallet_address):  
    return await self.frontend_service.build_swap_transaction(  
        params, wallet_address  
    )
```

```
async def get_pools(self):  
    return await self.frontend_service.get_all_pools()
```

```
async def get_pool_info(self, token_a, token_b):  
    return await self.frontend_service.get_pool_info(token_a, token_b)
```

```
class DEXAPI:
```

```
    def __init__(self, dex):  
        self.dex = dex  
        self.app = web.Application()  
        self._setup_routes()
```

```
    def _setup_routes(self):
```

```
self.app.router.add_get("/quote", self.handle_quote)
self.app.router.add_post("/swap", self.handle_swap)
self.app.router.add_get("/pools", self.handle_pools)
self.app.router.add_get("/pool/{token_a}/{token_b}",
self.handle_pool)
self.app.router.add_post("/liquidity/add", self.handle_add_liquidity)
self.app.router.add_get("/stats", self.handle_stats)
```

```
async def handle_quote(self, request):
    token_in = request.query.get("tokenIn")
    token_out = request.query.get("tokenOut")
    amount = int(request.query.get("amount", 0))
```

```
quote = await self.dex.get_quote(token_in, token_out, amount)
```

```
return web.json_response(quote)
```

```
async def handle_swap(self, request):
    data = await request.json()
```

```
tx = await self.dex.build_swap_tx(data, data["wallet"])
```

```
return web.json_response({
    "to": tx["to"],
    "data": tx["data"].hex() if isinstance(tx["data"], bytes) else
tx["data"],
    "value": str(tx["value"]),
    "gas": str(tx["gas"])
})
```

```
async def handle_pools(self, request):
    pools = await self.dex.get_pools()
```

```
return web.json_response({"pools": pools})
```

```
async def handle_pool(self, request):
    token_a = request.match_info["token_a"]
    token_b = request.match_info["token_b"]
```

```
pool = await self.dex.get_pool_info(token_a, token_b)
```

```

return web.json_response(pool)

async def handle_add_liquidity(self, request):
    data = await request.json()

    quote = await self.dex.frontend_service.get_liquidity_quote(
        data["token_a"],
        data["token_b"],
        int(data["amount_a"]))
    )

    return web.json_response(quote)

async def handle_stats(self, request):
    pools = await self.dex.get_pools()

    total_tvl = sum(p.get("reserve0", 0) + p.get("reserve1", 0) for p in
pools)
    total_pairs = len(pools)

    return web.json_response({
        "total_pairs": total_pairs,
        "total_tvl": total_tvl,
        "total_volume_24h": 0
    })

def run(self, host="0.0.0.0", port=8080):
    web.run_app(self.app, host=host, port=port)

```

CHAPTER SUMMARY

This chapter covered building a complete DEX:

Pair contracts hold reserves, execute swaps, and manage LP tokens with the constant product formula.

Factory contracts create new trading pairs and maintain the pair registry.

Router contracts provide user-friendly swap and liquidity functions with deadline and slippage protection.

Event indexers track all DEX activity for analytics and UI data.

Frontend services build transactions and provide quotes for user interfaces.

Deployment involves deploying factory and router, then creating initial pairs.

Complete systems integrate all components with APIs for external access.

BOOK 6 SUMMARY

This book covered Advanced DEX Development:

Chapter 1: DEX Architecture - Centralized vs decentralized, AMM models, factory/router pattern.

Chapter 2: Liquidity Pool Mechanics - Constant product math, LP tokens, impermanent loss, fees.

Chapter 3: Automated Market Makers - Constant product, constant sum, StableSwap, weighted pools, concentrated liquidity.

Chapter 4: Price Discovery and Oracles - TWAP, Chainlink, manipulation resistance.

Chapter 5: Flash Loans and Arbitrage - Instant loans, arbitrage strategies, MEV, liquidations.

Chapter 6: Advanced Swap Routing - Pathfinding, split routing, multi-DEX routing.

Chapter 7: DEX Aggregation - Liquidity sources, price discovery, routing algorithms.

Chapter 8: Building Your Own DEX - Complete implementation from contracts to API.

Next book: Security and Auditing. We protect smart contracts from attacks.